Cold Boot Attacks on Post-Quantum Schemes

Ricardo Luis Villanueva Polanco

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
School of Mathematics and Information Security
Royal Holloway, University of London

2018

Declaration

These doctoral studies were conducted under the supervision of Professor Kenneth G. Paterson.

The work presented in this thesis is the result of original research I conducted, in collaboration with others, whilst enrolled in the School of Mathematics and Information Security as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Ricardo Luis Villanueva Polanco

June 2018

Abstract

Cryptographic models are intended to represent an adversary's capabilities when attacking encryption schemes. Models often err on the side of caution by over-estimating the power of adversaries. However, several recent attacks reported in the literature demonstrate that measuring an adversary's potential is a difficult task. This thesis will view the cryptographic landscape from the perspective of an adversary and the implementer.

We study how an adversary can take advantage of leaked information about a private key. The particular scenario we study is the cold boot attack whereby an adversary can procure a noisy version of the key (i.e. the extracted data will contain errors) from a computer's main memory. Such an attack is not traditionally modelled by the standard security games. We show how the adversary might recover the original secret key, and hence compromise security, for some lattice-based schemes such as NTRU and BLISS, as well as the signature scheme Rainbow, which is based on multivariate polynomials over a finite field, and finally the McEliece crypto-system, which is a code-based asymmetric encryption scheme.

We mount our attacks against specific real-world implementations of each of these schemes. For each scheme, we will study it and review at least one real-world implementation of the scheme. Moreover, for each implementation of a particular scheme, we will concern ourselves with acquiring knowledge of and evaluating each of the formats used to store the scheme's private key in memory, and then propose specific algorithms for key recovery in the cold boot attack setting.

Our approach to key recovery is general and based on the combination of key enumeration algorithms and other techniques. Basically, an original secret key is seen as a concatenation of multiple chunks, each of which has a fixed number of bits and takes multiple values. These chunks then are combined to produce candidates for the secret key. These key enumeration algorithms have been already used in other side-channel scenarios with a variety of different approaches being used to solving the problem.

Contents

1 Introduction **13**

1.1 Motivation . . . 13

1.2 Thesis Structure . . . 15

1.3 Associated Publications . . . 16

2 Background **17**

2.1 Side Channel Attacks . . . 17

2.2 Cold Boot Attacks . . . 18

2.2.1 RSA Setting . . . 21

2.2.2 Discrete Logarithm Setting . . . 22

2.2.3 Symmetric Key Setting . . . 22

2.2.4 Learning with Errors Setting . . . 22

2.3 Cold boot Attack Model . . . 23

2.4 Log Likelihood Statistic for Key Candidates . . . 24

2.5 Combining Chunks to Build Key Candidates . . . 24

3 Key Enumeration Algorithms **26**

3.1 Introduction . . . 26

3.1.1 Some Definitions . . . 26

3.1.2 Problem Statement . . . 27

3.2 Key Enumeration Algorithms . . . 29

3.2.1 An Optimal Key Enumeration Algorithm . . . 29

3.2.1.1 Setup . . . 29

3.2.1.2 Basic Algorithm . . . 30

3.2.1.3 Complete Algorithm . . . 31

3.2.1.4 Memory Consumption . . . 34

3.2.2 A Bounded-Space Near-Optimal Key Enumeration Algorithm . . . 36

3.2.2.1 Basic Algorithm . . . 36

3.2.2.2 Complete Algorithm . . . 38
3.2.2.3 Parallelisation . . . 40
3.2.2.4 Variant . . . 42
3.2.3 A Simple Stack-Based, Depth-First Key Enumeration Algorithm . . . 42
3.2.3.1 Setup . . . 43
3.2.3.2 Complete Algorithm . . . 44
3.2.3.3 Speeding up the Pruning Process . . . 44
3.2.3.4 Memory Consumption . . . 45
3.2.3.5 Parallelisation . . . 46
3.2.3.6 `Threshold` Algorithm . . . 47
3.2.4 A Score-Based Key Enumeration Algorithm . . . 48
3.2.4.1 Complete Algorithm . . . 50
3.2.4.2 Parallelisation . . . 52
3.2.4.3 Running Times . . . 53
3.2.4.4 Memory Consumption . . . 54
3.2.5 A Key Enumeration Algorithm using Histograms . . . 56
3.2.5.1 Setup . . . 56
3.2.5.2 Complete Algorithm . . . 57
3.2.5.3 Parallelisation . . . 59
3.2.5.4 Memory Consumption . . . 60
3.2.5.5 Equivalence with the path counting approach . . . 61
3.2.6 A Quantum Key Search Algorithm . . . 62
3.3 Comparison of Key Enumeration Algorithms . . . 66
3.3.1 Implementation . . . 66
3.3.2 Scenario . . . 66
3.3.3 Results per algorithms . . . 67
3.3.4 Discussion . . . 71
3.4 Chapter Conclusions . . . 72

4 Cold Boot Attack on NTRU **74**
4.1 Introduction . . . 74
4.2 NTRU Encryption Scheme . . . 76
4.2.1 NTRU Public Key Encryption Scheme . . . 77
4.2.2 The NTRU Key Recovery Problem . . . 78
4.3 Private Key Formats for NTRU Implementations . . . 78

4.3.1 The tbuktu/Bouncy Castle Java Implementation. 79
4.3.2 Reference Parameters for tbuktu 80
4.3.3 The ntru-crypto Java Implementation 80
4.3.4 The ntru-crypto C Implementation 80
4.3.5 Reference Parameters for ntru-crypto 81
4.4 Mounting Cold Boot Key Recovery Attacks 82
4.4.1 Key Recovery Strategy 82
4.4.2 The ntru-crypto Java Implementation 83
4.4.3 The ntru-crypto C Implementation 84
4.4.4 The tbuktu Java Implementation 86
4.5 Experimental Evaluation 87
4.5.1 Implementation 87
4.5.2 Parallelisation 88
4.5.3 Search Intervals 88
4.5.4 Simulations 89
4.5.5 Results for the ntru-crypto Java Implementation 89
4.5.6 Results for the tbuktu Java Implementation 90
4.5.6.1 Counting Candidates and Estimating Running Times . . . 90
4.5.6.2 Parameters 91
4.5.6.3 Results – Complete Enumeration 91
4.5.6.4 Results – Partial Enumeration 93
4.5.6.5 Running times 94
4.6 Chapter Conclusions 94

5 Cold Boot Attacks on BLISS **95**
5.1 Introduction 95
5.2 Preliminaries 97
5.2.1 Notation 97
5.2.2 Lattices and Bases 97
5.2.2.1 Shortest Vector Problem (SVP) 98
5.2.2.2 Bounded Distance Decoding (BDD) 98
5.2.2.3 Babai's Nearest Plane Algorithm 99
5.3 BLISS Signature Scheme 99
5.3.1 The BLISS Key Generation Algorithm 99
5.4 The strongSwan Project 100

5.4.1 The strongSwan BLISS Implementation . . . 100
5.5 Mounting Cold Boot Key Recovery Attacks . . . 102
5.5.1 Initial Observations . . . 102
5.5.2 Key Recovery Via Key Enumeration . . . 103
5.5.2.1 Constructing L_f from $\mathbf{s}'_1$ . . . 103
5.5.2.2 Constructing L_h from $\mathbf{s}'_2$ . . . 104
5.5.2.3 Combining L_f and L_h . . . 106
5.5.2.4 Enumerating Only Candidates for $\mathbf{f}$ . . . 107
5.5.3 Casting the Problem as an LWE Instance . . . 109
5.5.3.1 Meet-in-the-Middle Attacks . . . 111
5.5.3.2 Parallel Collision Search . . . 113
5.5.3.3 Hybrid attack on LWE . . . 115
5.5.3.4 Combining Lattice Techniques and Key Enumeration . . . 116
5.6 Experimental Evaluation . . . 118
5.6.1 Simulations . . . 118
5.6.2 Key Recovery Via Key Enumeration . . . 119
5.6.2.1 Parameters . . . 119
5.6.2.2 Setup . . . 119
5.6.2.3 Key Enumeration Algorithm for Phase II . . . 119
5.6.2.4 Results . . . 121
5.7 Chapter Conclusions . . . 124

6 Cold Boot Attacks on Rainbow **126**
6.1 Introduction . . . 126
6.2 Multivariable Cryptosystems . . . 128
6.2.1 The Oil and Vinegar Signature Scheme . . . 129
6.2.2 Rainbow, a Signature Scheme . . . 131
6.3 Rainbow Implementations . . . 134
6.3.1 The Reference Implementation . . . 134
6.3.2 The Bouncy Castle Implementation . . . 138
6.4 Mounting Cold Boot Attacks . . . 141
6.4.1 Key Recovery Algorithm . . . 141
6.4.1.1 Constructing High Scoring Candidates for an Affine Map . 143
6.4.1.2 Constructing High Scoring Candidates for f. . . . 147
6.4.1.3 Memory Consumption . . . 150

6.4.1.4 Running Time . . . 151
6.4.1.5 Parallelisation . . . 152
6.5 Experimental Evaluation . . . 152
6.5.1 Implementation . . . 152
6.5.2 Setup . . . 153
6.5.3 Success Rate . . . 154
6.6 Chapter Conclusions . . . 157

7 Cold Boot Attacks on McEliece Public Key Encryption Scheme 159
7.1 Introduction . . . 159
7.2 Goppa Codes . . . 161
7.2.1 Parameters of a Goppa Code . . . 161
7.2.1.1 Parameters for a special case . . . 162
7.2.2 Parity Check Matrix of a Goppa Code . . . 162
7.2.3 Generator Matrix of a Goppa Code . . . 164
7.2.4 Correcting Errors . . . 164
7.2.4.1 Correcting Errors in a Special case . . . 166
7.2.5 Decoding a Message . . . 168
7.3 McEliece Public Key Encryption Scheme . . . 169
7.3.1 Public Key Encryption Scheme . . . 169
7.3.1.1 Key Generation . . . 169
7.3.1.2 Encryption . . . 170
7.3.1.3 Decryption . . . 170
7.4 Software Implementations for McEliece Public Key Encryption Scheme . . . 170
7.4.1 `Bouncy Castle` Implementation . . . 170
7.5 Mounting Cold Boot Attacks . . . 175
7.5.1 Cold Boot Attack Model . . . 175
7.5.2 Key Recovery . . . 176
7.5.3 Key Recovery Algorithm . . . 177
7.5.4 Constructing Array Candidates . . . 179
7.5.4.1 Constructing List of Candidates for Irreducible Polynomials . . . 179
7.5.4.2 Constructing List of Candidates for a 2^m-permutation . . . 181
7.5.5 Parallelisation . . . 182
7.6 Experimental Evaluation . . . 182

7.6.1 Implementation 182
7.6.2 Setup 183
7.6.3 Success Rate 184
7.7 Chapter Conclusions 187

8 Concluding Remarks **188**

Bibliography **191**

List of Figures

3.1 Binary tree built from L^0, L^1, L^2, L^3, L^4. 32

3.2 Geometric representation of the key space divided into layers of width $\omega = 3$. 37

3.3 Geometric representation of the key enumeration within $layer_3^3$. 38

3.4 Geometric representation of the key enumeration by variant. 42

3.5 Running Times of Algorithm 13 of KEA with histograms from Section 3.2.5,. The y-axis represents the running time (milliseconds), while the x-axis represents N_b. 68

4.1 Success rate of our algorithm (y-axis) against β (x-axis) for a fixed $\alpha = 0.001$, using Class II intervals. 90

4.2 Expected success rate for a full enumeration for $\alpha = 0.001, 0.09$. The y-axis represents the success rate, while the x-axis represents β. 92

4.3 Success rate for enumeration with 2^{40} keys over a Class I interval for $\alpha = 0.001, 0.09$, for different values of `lsize`. The y-axis represents the success rate, while the x-axis represents β. 93

4.4 Success rates for full enumeration, and partial enumeration with 2^{30} keys, 2^{40} keys for $\alpha = 0.001, 0.09$ and with `lsize` $= 1024$. The y-axis represents the success rate, while the x-axis represents β. 93

5.1 struct `private_bliss_private_key_t` 101

5.2 Expected success rate for a full enumeration for $\alpha = 0.001$. The y-axis represents the success rate, while the x-axis represents β. 122

5.3 Expected success rate for a 2^{40} enumeration for $\alpha = 0.001$. The y-axis represents the success rate, while the x-axis represents β. 123

5.4 Expected success rate for various enumerations for $\alpha = 0.001$ and `blsize` $=$ 1024. The y-axis represents the success rate, while the x-axis represents β. 123

7.1 Key Generation Algorithm of `Bouncy Castle` Implementation. 172

List of Tables

3.1 Variation of the number of key candidates in KEA with histograms (3.2.5). 69
3.2 Number of bits for storing histograms in KEA with histograms (3.2.5). . . . 70
3.3 Qualitative and functional attributes of key enumeration algorithms. 72

4.1 Format of private key blob in `ntru-crypto` C Implementation 81

6.1 Values for $p_{s_i}, 1 \leq i \leq 3$, for the ad hoc parameter set. 156

7.1 Values for p_1 and p_2 for the ad hoc parameter set, with $\texttt{blsize}_1 = \texttt{blsize}_2 = 2^9$. 185
7.2 Values for p_1 and p_2 for the ad hoc parameter set, with $\texttt{blsize}_1 = \texttt{blsize}_2 = 2^{10}$. 186

Acknowledgements

First and foremost, I would like to sincerely thank my supervisor, Kenny Paterson, for tolerating me for all these years. His support and guidance were invaluable, and despite being incredibly busy, he would always take the time to help me when it mattered most.

I am very grateful for the financial support of Colciencias. Without it, my Ph.D. journey would unlikely have started in the first place. I also would like to thank my wife Karina Jimenez and our son Samuel David for their love through these years. Also, many thanks to my parents, Ricardo and Doris, and brother Samir for their continuous support and motivation.

Chapter 1

Introduction

Contents

1.1 Motivation 13
1.2 Thesis Structure 15
1.3 Associated Publications 16

In this chapter we provide motivation for this thesis, and we provide a roadmap of the results we shall present.

1.1 Motivation

The research on cryptographic algorithms has been developing gradually in recent decades, evolving from an application of human creative skill and imagination to a systematic study of the structure and behaviour of such cryptographic algorithms. During this time there has been a gradual divergence between the theoretical and practical cryptographic aspects found in the literature. A serious and considerable difference materialises when attempting to evaluate or estimate the nature, abilities, influence or power associated with an adversary. When designers attempt to design a cryptographic scheme, these designers commence modelling the powers or abilities that an adversary may have to attack their scheme, and then demonstrate their scheme is secure against any adversary having the modelled powers. However, these models may not suffice because there is an intrinsic problem with them, i.e., the designers may not be able to accurately assess all possible cunning actions an adversary can perform with the purpose of undermining their cryptographic algorithms. As a consequence of this issue, various schemes have been proved to be secure according to certain models, but these schemes may be easily broken by attacks that fall out of the scope of these models. Such unexpected failures have taken place in the real world frequently [14, 42, 46, 64, 73, 74].

In this thesis we will study algorithms by which an adversary might reconstruct a private key of a particular cryptographic scheme when the adversary is able to obtain a leakage function of the private key. Traditional security models do not allow an adversary to have access to such deliberate disclosure of confidential information, and hence various cryptographic schemes may become insecure when an adversary is given access to such undisclosed information. In practice such information could be obtained, although with some effort on the attacker's part, by gathering data leaked from the physical effects caused by the functioning of the cryptographic scheme's implementation [46]. In particular, the adversary could procure data from a computer's main memory via a cold boot attack [28]. This attack's idea originates in taking advantage of the fact that modern computer memories continue to have data for a portion of time after the computer's power has been interrupted. Hence, an attacker with physical access to a machine might be able to recover, for example, cryptographic key information as a result of this effect. Unluckily for such an adversary, after the computer's power has been interrupted, the bits in memory will be subject to a gradual degradation. Therefore, any data extracted by an adversary from the computer's main memory will probably be altered in a number of random bits, i.e. any retrieved data will likely differ from the original data. So the logical question that becomes apparent is: Once an attacker has obtained a noisy version of a cryptographic key, is it possible for the attacker to reconstruct the original key?

This question has already been addressed for broad classes of classical cryptographic schemes, both symmetric and asymmetric, so we will instead focus on that question for several post-quantum cryptographic schemes. Our work can be seen as a continuation of the tendency towards developing cold boot attacks for different schemes. But it can also be seen as the beginning of the evaluation of the leading post-quantum candidates against this class of attack. Such an evaluation should form a small but important part of the overall assessment of schemes in the NIST selection process for post-quantum algorithms. We concern ourselves with the reconstruction of private keys of some lattice-based schemes, such as NTRU and BLISS, as well as the signature scheme Rainbow, which is based on multivariate polynomials over a finite field, and the McEliece crypto-system, which is a code-based asymmetric encryption scheme. Our study is aimed at specific implementations of each of the cryptographic schemes. Indeed, for each of the schemes, we will review at least a real implementation of the scheme with the purpose of learning on its in-memory formats for the scheme's private key and then propose specific algorithms for key recovery in the cold boot attack setting.

1.2 Thesis Structure

Chapter 2 This chapter includes a brief background on some important concepts and definitions in side-channel attacks, particularly those regarding cold boot attacks, that we use throughout this thesis.

Chapter 3 In this chapter, we investigate the key enumeration problem. We give a detailed description of some existing key enumeration algorithms and propose variants for some of them. Furthermore, we make a comparison of the most important qualitative features of these key enumeration algorithms.

Chapter 4 This chapter analyses the feasibility of cold boot attacks against the NTRU public key encryption scheme. Our approach is first to review two specific implementations to learn about the formats these implementations use to store the NTRU private key in memory. The first implementation we review is `ntru-crypto`, which is a pair of C and Java libraries developed by OnBoard Security. The second is `tbuktu`, which is also available in C and Java languages. We propose a general key recovery strategy that is adapted to each of the in-memory private key formats.

Chapter 5 This chapter studies the practicability of cold boot attacks against the BLISS signature scheme. We review an implementation provided by the `strongSwan` project to learn about the formats this implementation uses to store the BLISS private key. We then propose a key recovery strategy based on combining key enumeration algorithms to generate possible key candidates. Additionally, we establish a connection between the key recovery problem for this particular case and a non-conventional instance of the Learning with Errors Problem (LWE). We then explore other techniques based on the meet-in-the-middle generic attack and lattices to tackle this instance of LWE.

Chapter 6 This chapter studies the achievability of cold boot attacks against the Rainbow signature scheme. By reviewing the reference implementation and the implementation provided by the `Bouncy Castle` project, we learn about the in-memory formats these implementations use to store a Rainbow private key. We then propose a key recovery strategy

that exploits the structure of this signature scheme, allowing us to split the components of a Rainbow private key into chunks. We then make use of key enumeration algorithms to generate feasible candidates for each of these components and then proceed to combine these candidates to obtain possible key candidates.

Chapter 7 In this chapter, we study the McEliece public key encryption scheme in the cold boot attack setting. In particular, we focus on an implementation provided by the `Bouncy Castle` project. For this implementation, we study its formats for storing a private key for this scheme and propose a key recovery strategy that combines key enumeration techniques with some linear algebra algorithms.

1.3 Associated Publications

In the course of my PhD, I have been fortunate to work on some diverse projects and some results from them have already been published.

1. K. G. Paterson, R. Villanueva-Polanco R. Cold Boot Attacks on NTRU. In A. Patra, N. Smart, editors, *INDOCRYPT 2017*, volume 10698 of *LNCS*, pages 107–125, Springer, Nov. 2017.

Chapter 2

Background

Contents

2.1 Side Channel Attacks 17
2.2 Cold Boot Attacks 18
2.3 Cold boot Attack Model 23
2.4 Log Likelihood Statistic for Key Candidates 24
2.5 Combining Chunks to Build Key Candidates 24

In this chapter, we will include a brief background on some important concepts in side channel attacks, particularly those concepts regarding cold boot attacks. We will also give an overview about previous works carried out on cold boot attacks for some classical cryptographic schemes.

2.1 Side Channel Attacks

A side-channel attack may be defined as any attack based on information obtained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself. When performing such attacks, an adversary procure extra information from the implementation by exploiting any extra source of information, such as timing information, power consumption, electromagnetic leaks or even sound, with the intention of subverting cryptographic algorithms. There are various types of side channel attacks. For example, *cache attacks*, *timing attacks* and *data remanence attacks*. When carrying out a cache attack, the attacker has the capability to observe and check the progress of cache accesses made by the victim over a period of time in a shared physical system as in a virtualised environment or a type of cloud service [6, 13, 73, 74]. On the other hand,

when carrying out a timing attack, the attacker has the capability to take measurements of how much time various computations (such as, say, comparing an attacker's given password with the victim's unknown one) take to perform [14, 15, 42, 63, 64]. Finally, when performing a data remanence attack, the attacker is able to read sensitive data from a source of computer memory after supposedly having been deleted (e.g. cold boot attacks [28]). In all cases, the underlying principle is that physical effects caused by the operation of a cryptographic scheme's implementation (on the side) can provide useful extra information about secrets in the system, for example, the cryptographic key, partial state information, full or partial plain-texts and so forth [46].

2.2 Cold Boot Attacks

A cold boot attack is a type of data remanence attack by which sensitive data are read from a computer's main memory after supposedly having been deleted. This attack relies on the data remanence property of DRAM to retrieve memory contents that remain readable in the seconds to minutes after power has been removed. This attack was first described in the literature by Halderman *et al.* nearly a decade ago [28] and since then it has received significant attention. In this setting, an attacker with physical access to a computer is able to retrieve content from a running operating system after using a cold reboot to restart the machine. A running computer is cold-booted when the operating system is not shut down in an orderly manner, skipping file system synchronisation and other activities that would occur on an orderly shutdown. Therefore, after cold-rebooting the machine, such an attacker may use a removable disk to boot a lightweight operating system, which is then used to dump the contents of pre-boot physical memory to a file. Alternatively, such an attacker may remove the memory modules from the original computer and quickly place them in a compatible computer under the attacker's control, which is then booted to access the memory. Further analysis can then be performed against the data that was dumped from memory to find various sensitive information, such as cryptographic keys contained in it. This task may be performed by making use of various forms of key finding algorithms [67, 28]. Unfortunately for such an adversary, the bits in memory will experience a process of degradation once the computer's power is interrupted. This implies that if the adversary is able to retrieve any data from the computer's main memory after the power is cut off, the extracted data will probably have random bit fluctuations, i.e., the data will be noisy, or rather, be dissimilar from the original data.

Internally, each DRAM cell is essentially a capacitor, i.e., a device used to store an electric charge, consisting of one or more pairs of conductors separated by an insulator. So a cell encodes a single bit by keeping a charge in it. Over time, this charge will leak out of the capacitor and hence the cell will lose its state, i.e., it will decay to its "ground state", either 0 or 1, depending on how the capacitor is physically connected internally. To forestall this decay, the cells in a DRAM chip are refreshed periodically. So it is expected that after power is lost, each DRAM cell will retain its value for a while and then will decay to its "ground state". The amount of time for which cell values are maintained while the power is off depends on the particular memory type and the ambient temperature. In fact, [28] reports results of multiple experiments in which representative memory regions were filled with a pseudorandom pattern and then were read back after varying periods of time without refresh and under different temperature conditions. Such results reveal that, at normal operating temperatures ($25.5°C$ to $44.1°C$), there is little corruption within the first few seconds, but this phase is then followed by a rapid decay. However, the period of mild corruption can be prolonged by cooling the memory chips. For instance, according to [28], in an experiment at $-50°C$ (which can be achieved by spraying compressed air onto the memory chips) less than 0.1% of bits decay within the first minute. At temperatures of approximately $-196°C$ (achieved by means of the use of liquid nitrogen) less than 0.17% of bits decay within the first hour. Notably, once power has been switched off, the memory will be partitioned into regions, and each region will have a "ground state" which is associated with a bit, 0 or 1. In a 0 ground state, the 1 bits will eventually decay to 0 bits, while the probability of a 0 bit switching to a 1 bit is very small, but not vanishing (a common probability is circa 0.001 [28]). When the ground state is 1, the opposite is true.

Furthermore, a single bit of a charge-based memory (DRAM) may spontaneously flip to the opposite state, as a result of an electrical or magnetic interference inside a computer system. This is normally referred to as a soft (or transient) fault since only the data-value has been changed. Particularly, the majority of one-off errors in DRAM chips happen as a result of background radiation, which may change the contents of one or more memory cells or interfere with the circuitry used to read/write them [52]. This problem may be mitigated by using redundant memory bits and additional circuitry that use these bits to detect and correct soft errors. In most cases, the detection and correction are performed by the memory controller, which uses the extra memory bits to calculate parity bits, enabling missing data to be reconstructed by an error-correcting code (ECC) [52]. On the other

hand, there are also faults that are hard or permanent, referring to faulty cells that may not store any value reliably, as a result of manufacturing-time defects.

In practice, even though the content (with errors) of memory may be retained for a period of time via cooling techniques, an attacker has yet to extract such content from memory before even considering reconstructing any relevant information from it. To extract memory images, the attacker first has to take into account several possible issues that may arise. For instance, when the target machine boots again, the system BIOS may write portions of memory with its own code and data, although the affected portions normally are small. Also, in some machines, the BIOS may perform a destructive memory check during its Power-On Self Test (POST) (however this test might be disabled or bypassed in some machines). To handle these issues, the authors of [28] reported they used tiny special-purpose programs (memory-imaging tools) that, when booted from either a warm or cold reset state, produced accurate dumps of memory contents to some external medium. These special-purpose programs used trivial amounts of RAM, and their memory offsets were adjusted to some extent to ensure that data structures of interest were unaffected. Additionally, the authors of [28] pointed out that if an attacker cannot force a target system to boot memory-imaging tools, the attacker could physically remove the memory modules, place them in a computer selected by the attacker and then dump their contents. Note that removing the memory modules may allow the attacker to image memory regions where standards BIOSes (or memory-imaging tools) load their own code during boot.

Once the attacker extracts the memory content, the attacker has to profile this content to gain knowledge about the regions of memory and the probabilities of both a 1 flipping to 0 and a 0 flipping to 1. According to the results of the experiments reported in [28], almost all memory bits tend to decay to predictable ground states, with only a tiny fraction flipping in the opposite direction. Also, the authors pointed out that the probability of decaying to the "ground state" increases as time goes on, while the probability of flipping in the opposite direction remains relatively constant and tiny (e.g. 0.001). These results suggests that the attacker may model the decay in a region as a binary asymmetric channel, i.e., assuming the probability of a 1 flipping to 0 is some fixed number, while the probability of a 0 flipping to a 1 is some other fixed number.

In practice, the attacker can determine the "ground state" of a particular region of memory

rather easily in an attack by reading all the bits and determining how many of them are 0 bits and how many are 1 bits. Furthermore, the attacker may estimate the probabilities by comparing any original content in such region with its respective noisy version. Another significant challenge posed to the attacker once the attacker has obtained memory images is extracting encryption keys from them. The authors of [28] presented algorithms to locate AES keys and RSA keys in a memory image and, although such algorithms are scheme-specific, their intrinsic idea may be employed for other schemes, since it relies on determining identifying features of the formats for storing a scheme-specific key and using such identifying features as markers to identify sequences of bytes, while searching over decayed memory images. More specifically, the algorithm searches for sequences of bytes with low Hamming distance to these markers and checks that the remaining bytes in a candidate sequence satisfy some conditions.

Because only a noisy version of the original key may be retrievable from main memory once the attacker discovers the location of the data in it, the adversary's main task then becomes the mathematical problem of recovering the original key from a noisy version of that key. Additionally, the adversary may have access to reference cryptographic data created using that key (e.g. cipher-texts for a symmetric key encryption scheme) or have a public key available (in the asymmetric setting). So the focus of cold boot attacks after the initial work pointing out their feasibility [28] has been to develop algorithms for efficiently recovering keys from noisy versions of those keys for a range of different cryptographic schemes, whilst exploring the limits of how much noise can be tolerated.

2.2.1 RSA Setting

Heninger and Shacham [30] focussed on the case of RSA keys, giving an efficient algorithm based on Hensel lifting to exploit redundancy in the typical RSA private key format. This work was followed up by Henecka, May and Meurer [29] and Paterson, Polychroniadou and Sibborn [55], with both papers also focusing on the mathematically highly structured RSA setting. The latter paper in particular pointed out the *asymmetric* nature of the error channel intrinsic to the cold boot setting and recast the problem of key recovery for cold boot attacks in an information theoretic manner.

2.2.2 Discrete Logarithm Setting

Lee *et al.* [44] were the first that discussed these attacks in the discrete logarithm setting. Their attack model assumed that an attacker only had access to the public key g^x and a noisy version of the private key x. Additionally, their model made the assumption that an adversary knew an upper bound for the number of errors in the private key. Because this latter assumption might not be realistic and the attacker did not have access to further redundancy, their proposed algorithm would likely be unable to recover keys that were affected by particularly high noise levels in the true cold boot scenario, i.e., only assuming a bit-flipping model. This work was improved upon by Poettering and Sibborn [60]. In their paper, they attempted to determine whether there were any in-memory private key representations that contained redundancy that could be used to improve cold boot key-recovery algorithms, in practical software implementations. Thus, they considered two scenarios that were taken from two ECC implementations found in TLS libraries: the windowed non-adjacent form (wNAF) representation used in OpenSSL, and the comb-based approach used in PolarSSL. Through exploiting redundancies found in the respective in-memory private key representations, they developed cold boot key-recovery algorithms that were applicable to the true cold boot scenario.

2.2.3 Symmetric Key Setting

Other papers have considered cold boot attacks in the symmetric key setting, including Albrecht and Cid [2] who focused on the recovery of symmetric encryption keys in the cold boot setting by employing polynomial system solvers, and Kamal and Youssef [39] who applied SAT solvers to the same problem. Further research on the development of cold boot attacks for specific schemes can be found in [44, 37]. Cold boot attacks are also widely cited in the theoretically-oriented literature on leakage-resilient cryptography, but the relevance there is marginal because the cold boot attack scenario (direct access to a noisy version of the whole key) does not really apply in the leakage-resilient setting.

2.2.4 Learning with Errors Setting

A recent research paper by Albrecht *et al.* [4] explored cold boot attacks on cryptographic schemes based on the ring – and module – variants of the Learning with Errors (LWE)

problem. In particular, they considered two cryptographic schemes: the Kyber key encapsulation mechanism (KEM) and New Hope KEM. They also considered two encodings to store LWE keys. The first encoding stores polynomial coefficients directly in memory, while the second encoding performs a number theoretic transform (NTT) on the key before storing it. This method is commonly used to improve efficiency in implementations. They then developed an attack strategy exploiting the structure added by using an NTT on the key. In particular, their strategy exploited the property that a 2^n-dimensional Fourier transform can be written in terms of two 2^{n-1}-dimensional Fourier transform. So, using this halving property, they split their cold boot NTT decoding problem into two smaller cold boot NTT decoding problems, i.e. employing a divide and conquer approach. As a result, they showed that, at a 1% bit-flip rate, a cold boot attack on Kyber KEM parameters had a cost of 2^{43} operations when the NTT-based encoding is used for key storage, compared to 2^{70} operations with the first encoding.

2.3 Cold boot Attack Model

Our cold boot attack model assumes that the adversary can obtain a noisy version of a secret key (using whatever format is used to store it in memory). We assume that the corresponding public parameters are known exactly (without noise). We do not consider here the important problem of how to locate the appropriate area of memory in which the secret key bits are stored, though this would be an important consideration in practical attacks. Our aim is then recover the secret key. Note that it is sufficient to recover a list of key candidates in which the true secret key is located, since we can always test a candidate by executing known algorithms linked to the scheme we are attacking.

We assume throughout that a 0 bit of the original secret key will flip to a 1 with probability $\alpha = P(0 \to 1)$ and that a 1 bit of the original private key will flip with probability $\beta = P(1 \to 0)$. We do not assume that $\alpha = \beta$; indeed, in practice, one of these values may be very small (e.g. 0.001) and relatively stable over time, while the other increases over time. Furthermore, we assume that the attacker knows the values of α and β and that they are fixed across the region of memory in which the private key is located. These assumptions are reasonable in practice: one can estimate the error probabilities by looking at a region where the memory stores known values (e.g. where the public key is located), and the regions are typically large.

2.4 Log Likelihood Statistic for Key Candidates

Suppose we have a secret key that is W bits in size, and let $\mathtt{r} = (b_0, \ldots, b_{W-1})$ denote the bits of the noisy key (input to the adversary in the attack). Suppose a key recovery algorithm constructs a candidate for the private key $\mathtt{c} = (c_0, \ldots, c_{W-1})$ by some means (to be determined). Then, given the bit-flip probabilities α, β, we can assign a likelihood score to $\mathtt{c}$ as follows:

$$L[\mathtt{c}; \mathtt{r}] := \Pr[\mathtt{r}|\mathtt{c}] = (1-\alpha)^{n_{00}} \alpha^{n_{01}} \beta^{n_{10}} (1-\beta)^{n_{11}},$$

where n_{00} denotes the number of positions where both $\mathtt{c}$ and $\mathtt{r}$ contain a 0 bit, n_{01} denotes the number of positions where $\mathtt{c}$ contains a 0 bit and $\mathtt{r}$ contains a 1 bit, etc.

The method of maximum likelihood estimation[1] then suggests picking as $\mathtt{c}$ the value that maximises the above expression. It is more convenient to work with log likelihoods, and equivalently to maximise these, viz:

$$\mathcal{L}[\mathtt{c}; \mathtt{r}] := \log \Pr[\mathtt{r}|\mathtt{c}] = n_{00} \log(1-\alpha) + n_{01} \log \alpha + n_{10} \log \beta + n_{11} \log(1-\beta).$$

We will frequently refer to this log likelihood expression as a *score* and seek to maximise its value (or, equally well, minimise its negative).

2.5 Combining Chunks to Build Key Candidates

Let us suppose that the encoding of the true secret key $\mathtt{r}$ can be represented as a concatenation of W/w chunks, each on w bits. Let us name the chunks $\mathtt{r}^0, \mathtt{r}^1, \ldots, \mathtt{r}^{W/w-1}$ so that $\mathtt{r}^i = b_{i \cdot w} b_{i \cdot w+1} \ldots b_{i \cdot w+(w-1)}$. Additionally, let us suppose that key candidates $\mathtt{c}$ can also be represented by concatenations of chunks $\mathtt{c}^0, \mathtt{c}^1, \ldots, \mathtt{c}^{W/w-1}$ in the same way.

Suppose further that each of the at most 2^w candidate values for chunk $\mathtt{c}^i$ $(0 \leq i < W/w)$ can be enumerated and given its own score by some procedure (formally, a sub-algorithm in an overall attack). For example, the above expression for log likelihood across all W bits of secret key can be easily modified to produce a log likelihood expression for any

[1] See for example `https://en.wikipedia.org/wiki/Maximum_likelihood_estimation`.

candidate for chunk i as follows:

$$\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i] := \log \Pr[\mathtt{r}^i | \mathtt{c}^i] = n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta), \quad (2.1)$$

where the n^i_{ab} values count occurrences of bits across the i-th chunks, $\mathtt{r}^i$, $\mathtt{c}^i$.

We can therefore assume that we have access to W/w lists of scores, each list containing up to 2^w entries. The W/w scores, one from each of these per-chunk lists, can be added together to create a total score for a complete candidate $\mathtt{c}$. Indeed, this total score is statistically meaningful in the case where the per-chunk scores are log likelihoods because of the additive nature of the scoring function in that case. The question then becomes: can we devise efficient algorithms that traverse the lists of scores to combine chunk candidates $\mathtt{c}^i$, obtaining complete key candidates $\mathtt{c}$ having high total scores (with total scores obtained by summation)? This is a question that has been previously addressed in the side-channel analysis literature [69, 11, 48, 47, 18], with a variety of different algorithmic approaches being possible to solve the problem. We will explore these approaches in detail in the next chapter.

Chapter 3

Key Enumeration Algorithms

Contents

3.1 Introduction 26
3.2 Key Enumeration Algorithms 29
3.3 Comparison of Key Enumeration Algorithms 66
3.4 Chapter Conclusions 72

In this chapter, we detail and investigate key enumeration algorithms. We propose variants for some existing key enumeration algorithms and make a comparison of the most important features of these key enumeration algorithms.

3.1 Introduction

In this section, we will introduce the key enumeration problem.

3.1.1 Some Definitions

We define an array A as a data structure consisting of a finite sequence of values of a specified type, i.e., $A = [a_0, \ldots, a_{n_A-1}]$. The length of an array, n_A, is established when the array is created. After creation, its length is fixed. Each item in an array is called an element, and each element is accessed by its numerical index, i.e., $A[i] = a_i$, with $0 \leq i < n_A$. Let $A^0 = [a_0^0, \ldots, a_{n_0-1}^0]$ and $A^1 = [a_0^1, \ldots, a_{n_1-1}^1]$ be two arrays of elements of a specified type. The associative operation $||$ is defined as follows.

$$[a_0^0, \ldots, a_{n_0-1}^0] \,||\, [a_0^1, \ldots, a_{n_1-1}^1] = [a_0^0, \ldots, a_{n_0-1}^0, a_0^1, \ldots, a_{n_1-1}^1].$$

A list L is defined as a resizable array of elements of a specified type.[1] Given a list $L = [e_0, \ldots, e_{n_l-1}]$, this data structure supports the following methods.

- The method $L.\texttt{size}()$ returns the number of elements in this list, i.e., the value n_l.
- The method $L.\texttt{add}(e_{n_l})$ appends the specified element e_{n_l} to the end of this list, i.e., $L = [e_0, e_1, \ldots, e_{n_l}]$ after this method returns.
- The method $L.\texttt{get}(j)$, with $0 \leq j < L.\texttt{size}()$, returns the element at the specified position j in this list, i.e., e_j.
- The method $L.\texttt{clear}()$ removes all the elements from this list. The list will be empty after this method returns, i.e., $L = []$.

3.1.2 Problem Statement

Let us assume that we have an encoding of a secret key $\texttt{r}$ that is W bits in size and that $\texttt{r}$ can be represented as a concatenation of W/w chunks, each on w bits. Let us set $\mathcal{N} = W/w$ and name the chunks as $\texttt{r}^0, \texttt{r}^1, \ldots, \texttt{r}^{\mathcal{N}-1}$ with $\texttt{r}^i = b_{i \cdot w} b_{i \cdot w+1} \ldots b_{i \cdot w+(w-1)}$. Similarly, we suppose there is a key recovery algorithm that constructs a key candidate $\texttt{c} = (c_0, \ldots, c_{W-1})$ for the encoding of the secret key by some means and that these key candidates $\texttt{c}$ can be represented by concatenations of chunks $\texttt{c}^0, \texttt{c}^1, \ldots, \texttt{c}^{\mathcal{N}-1}$ in the same way.

Let f be a function such that for a given candidate $\texttt{c}$, the function outputs a real value $f(\texttt{c})$. This function f is called a scoring function. Moreover, if for a given $\texttt{c}$, we have that $f(\texttt{c}) = \sum_{i=0}^{\mathcal{N}-1} f'(\texttt{c}^i)$, for some other scoring function f', then f is called an additive scoring function. For instance, we may define an additive scoring function f from the log-likelihood expression introduced in Section 2.5.

Assuming we have access to an additive scoring function f and can enumerate each of the at most 2^w values for chunk $\texttt{c}^i$ $(0 \leq i < \mathcal{N})$ and give it its own score by using f', then we can have access to $\mathcal{N}$ lists of *chunk candidates*, where each list contains up to 2^w entries. A *chunk candidate* is defined as a 2-tuple of the form $(score, value)$, where the first component *score* is a positive real number (candidate score) while the second component *value* is an array of w-bit strings (candidate value).

[1] A table $\mathcal{T}$ is also a resizable array of elements of a specified type.

Let $L^i = [\mathtt{c}_0^i, \mathtt{c}_2^i, \ldots, \mathtt{c}_{m_i-1}^i]$ be the list of chunk candidates for chunk i, $0 < m_i \leq 2^w$. Let $\mathtt{c}_{j_0}^{i_0}, \ldots, \mathtt{c}_{j_n}^{i_n}$ be chunk candidates, $0 \leq i_0 < \cdots < i_n < \mathcal{N}$, $0 \leq j_i < m_i$. The function $\mathtt{combine}(\mathtt{c}_{j_0}^{i_0}, \ldots, \mathtt{c}_{j_n}^{i_n})$ returns a new chunk candidate $\mathtt{c}$ such that

$$\mathtt{c} = (\mathtt{c}_{j_0}^{i_0}.score + \cdots + \mathtt{c}_{j_n}^{i_n}.score, \mathtt{c}_{j_0}^{i_0}.value \mid\mid \cdots \mid\mid \mathtt{c}_{j_n}^{i_n}.value).$$

Note that when $i_0 = 0, i_1 = 1, \ldots, i_{\mathcal{N}-1} = \mathcal{N} - 1$, $\mathtt{c}$ will be a full key candidate. We will next state the key enumeration problem.

Definition 3.1.1 *The key enumeration problem entails traversing the $\mathcal{N}$ lists L^i, $0 \leq i \leq \mathcal{N}-1$, while picking a chunk candidate $\mathtt{c}_{j_i}^i$ from each L^i to generate full key candidates $\mathtt{c} = \mathtt{combine}(\mathtt{c}_{j_0}^0, \ldots, \mathtt{c}_{j_{\mathcal{N}-1}}^{\mathcal{N}-1})$. Moreover, we call an algorithm generating full key candidates $\mathtt{c}$ a key enumeration algorithm (KEA).*

Note that the key enumeration problem has been stated in a general way, however there are many other variants of this problem. These variants relate to the manner in which the key candidates are generated by a key enumeration algorithm.

A variant consists in enumerating key candidates $\mathtt{c}$ such that their total accumulated scores follow a specific order. For example, in many side-channel scenarios it is desirable to enumerate key candidates $\mathtt{c}$ starting at the one having the highest score, followed by the one having the second highest score and so on. In these scenarios, we need a key enumeration algorithm to enumerate high scoring key candidates in decreasing order based on their total accumulated scores. For example, such an algorithm would allow us to find the top M highest scoring candidates in decreasing order, where $1 \leq M \ll 2^W$. Furthermore, such an algorithm is known as an optimal key enumeration algorithm.

Another variant consists in enumerating all the key candidates $\mathtt{c}$ such that their total accumulated scores satisfy a defined condition rather than a specific order. For example, we may need to enumerate all key candidates whose total accumulated scores lie in an interval $[B_1, B_2]$. In this scenario, we need a key enumeration algorithm to enumerate key candidates whose total accumulated scores lie in that interval. Such an algorithm may not enumerate all the key candidates in a decreasing order, still it does need to ensure that all of them will be generated once it has completed. This is, the algorithm only concerns itself with generating all the key candidates whose total accumulated scores satisfy the

condition in any order. Such an algorithm would allow us to find the top M highest scoring candidates in any order if the interval is well-defined, for example. Moreover, such an algorithm is commonly known as a non-optimal key enumeration algorithm.

Broadly speaking, optimal key enumeration algorithms [68, 69] tend to consume more memory and be less efficient while generating high scoring key candidates, whereas non-optimal key enumeration algorithms [11, 48, 47, 61, 18] are expected to run faster and consume less memory.

3.2 Key Enumeration Algorithms

We will next present several key enumeration algorithms. These algorithms will be detailed and analysed below.

3.2.1 An Optimal Key Enumeration Algorithm

We study the optimal key enumeration algorithm (OKEA) that was introduced in [69]. We will firstly give the basic idea behind the algorithm by assuming the encoding of the secret key is represented as two chunks, hence we have access to two lists of chunk candidates.

3.2.1.1 Setup

Let $L^0 = [\mathsf{c}_0^0, \mathsf{c}_1^0, \ldots, \mathsf{c}_{m_0-1}^0]$ and $L^1 = [\mathsf{c}_0^1, \mathsf{c}_1^1, \ldots, \mathsf{c}_{m_1-1}^1]$ be the two lists respectively. Each list is in decreasing order based on the score component of its chunk candidates.

Let us define an extended candidate as a 4-tuple of the form $C := (\mathsf{c}_{j_0}^0, \mathsf{c}_{j_1}^1, j_0, j_1)$ and its score as $\mathsf{c}_{j_0}^0.score + \mathsf{c}_{j_1}^1.score$. Additionally, let `Q` be a priority queue [17] that will store extended candidates in decreasing order based on their score.

This data structure `Q` supports three methods. Firstly, the method `Q.poll()` retrieves and removes the head from this queue `Q` or returns `null` if this queue is empty. Secondly, the method `Q.add`(e) inserts the specified element e into the priority queue `Q`. Thirdly, the method `Q.clear()` removes all the elements from the queue `Q`. The queue will be empty after this method returns. By making use of a heap, we can support any priority-queue

operation on a set of size n in $\mathcal{O}(\log_2(n))$ time [17].

Furthermore, let `X,Y` be two vectors of bits that grow as needed. These are employed to track an extended candidate C in `Q`. C is in `Q` if only if both $\mathtt{X}_{j_0}$ and $\mathtt{Y}_{j_1}$ are set to 1. By default, all bits in a vector initially have the value 0.

Algorithm 1 outputs the next highest-scoring key candidate from L^0,L^1.

```
1: function nextCandidate()
2:     (c^0_{j0}, c^1_{j1}, j0, j1) ← Q.poll();
3:     X_{j0} ← 0;Y_{j1} ← 0;
4:     if (j0 + 1) < L^0.size() and X_{j0+1} is set to 0 then
5:         c^0_{j0+1} ← L^0.get( j0 + 1);
6:         Q.add((c^0_{j0+1}, c^1_{j1}, j0 + 1, j1));
7:         X_{j0+1} ← 1;Y_{j1} ← 1;
8:     end if
9:     if (j1 + 1) < L^1.size() and Y_{j1+1} is set to 0 then
10:        c^1_{j1+1} ← L^1.get(j1 + 1);
11:        Q.add((c^0_{j0}, c^1_{j1+1}, j0, j1 + 1));
12:        X_{j0} ← 1;Y_{j1+1} ← 1;
13:    end if
14:    return c_{j0,j1} = combine(c^0_{j0}, c^1_{j1}); (the next key candidate).
15: end function
```

3.2.1.2 Basic Algorithm

At the initial stage, the queue `Q` will be created. Next the extended candidate $(\mathtt{c}_0^0, \mathtt{c}_0^1, 0, 0)$ will be inserted into the priority queue and both $\mathtt{X}_0$ and $\mathtt{Y}_0$ will be set to 1. In order to generate a new key candidate, the routine `nextCandidate`, defined in Algorithm 1, should be executed.

Let us assume that $m_0, m_1 > 1$. First the extended candidate $(\mathtt{c}_0^0, \mathtt{c}_0^1, 0, 0)$ will be retrieved and removed from `Q` and then $\mathtt{X}_0$ and $\mathtt{Y}_0$ will be set to 0. The two **if** blocks of instructions will then be executed, meaning that the extended candidates $(\mathtt{c}_1^0, \mathtt{c}_0^1, 1, 0), (\mathtt{c}_0^0, \mathtt{c}_1^1, 0, 1)$ will be inserted into `Q`. Moreover, the entries $\mathtt{X}_0, \mathtt{X}_1, \mathtt{Y}_0, \mathtt{Y}_1$ will be set to 1, while the other entries of `X` and `Y` will remain as 0. The routine `nextCandidate` will then return $\mathtt{c}_{0,0} = \mathtt{combine}(\mathtt{c}_0^0, \mathtt{c}_0^1)$, which is the highest score key candidate, since L^0 and L^1 are in decreasing order.

At this point, the two extended candidates $(\mathtt{c}_1^0, \mathtt{c}_0^1, 1, 0), (\mathtt{c}_0^0, \mathtt{c}_1^1, 0, 1)$ (both in `Q`) are the only ones that can have the second highest score. Therefore, if Algorithm 1 is called again, the first instruction will retrieve and remove the extended candidate with the second highest

score, say $(\mathtt{c}_0^0, \mathtt{c}_1^1, 0, 1)$, from $\mathtt{Q}$ and then the second instruction will set $\mathtt{X}_0$ and $\mathtt{Y}_1$ to 0. The first **if** condition will be attempted, but this time it will be false since $\mathtt{X}_1$ is set to 1. However, the second **if** condition will be satisfied and therefore $(\mathtt{c}_0^0, \mathtt{c}_2^1, 0, 2)$ will be inserted into $\mathtt{Q}$ and the entries $\mathtt{X}_0$ and $\mathtt{Y}_2$ will be set to 1. The method will then return $\mathtt{c}_{0,1} = \mathtt{combine}(\mathtt{c}_0^0, \mathtt{c}_1^1)$, which is the second highest score key candidate.

At this point, the two extended candidates $(\mathtt{c}_1^0, \mathtt{c}_0^1, 1, 0), (\mathtt{c}_0^0, \mathtt{c}_2^1, 0, 2)$ (both in $\mathtt{Q}$) are the only ones that can have the third highest score. To see why, we know that the algorithm has generated $\mathtt{c}_{0,0}$, $\mathtt{c}_{0,1}$ so far. Since L^0 and L^1 are in decreasing order, we have that either $\mathtt{c}_{0,0}.score \geq \mathtt{c}_{0,1}.score \geq \mathtt{c}_{1,0}.score \geq \mathtt{c}_{0,2}.score$ or $\mathtt{c}_{0,0}.score \geq \mathtt{c}_{0,1}.score \geq \mathtt{c}_{0,2}.score \geq \mathtt{c}_{1,0}.score$. Also, any other extended candidate yet to be inserted into $\mathtt{Q}$ cannot have the third highest score, for the same reason. Consider for example $(\mathtt{c}_1^0, \mathtt{c}_1^1, 1, 1)$: this extended candidate will be inserted into $\mathtt{Q}$ only if $(\mathtt{c}_1^0, \mathtt{c}_0^1, 1, 0)$ has been retrieved and removed from $\mathtt{Q}$. Therefore, if Algorithm 1 is executed again, it will return the third highest scoring key candidate and have the extended candidate with the fourth highest score placed at the head of $\mathtt{Q}$. In general, the manner in which this algorithm travels through the $m_0 \times m_1$ matrix of key candidates guarantees to output key candidates in a decreasing order based on their total accumulated score, i.e., this algorithm is an optimal key enumeration algorithm.

Regarding how fast the queue $\mathtt{Q}$ grows, let $N_{\mathtt{Q}}^s$ be the number of extended candidates in $\mathtt{Q}$ after the function `nextCandidate` has been called $s \geq 0$ times. Clearly, we have that $N_{\mathtt{Q}}^0 = 1$, since $\mathtt{Q}$ only contains the extended candidate $(\mathtt{c}_0^0, \mathtt{c}_0^1, 0, 0)$ after initialisation. Also, $N_{\mathtt{Q}}^{m_1 \cdot m_2} = 0$, because after $m_1 \cdot m_2$ calls to the function, there will be no more key candidates to be enumerated. Note that during the execution of the function `nextCandidate`, an extended candidate will be removed from $\mathtt{Q}$ and two new extended candidates might be inserted into $\mathtt{Q}$. Considering the way in which an extended candidate is inserted into the queue, $\mathtt{Q}$ may contain at most one element in each row and column at any stage, hence $N_{\mathtt{Q}}^s \leq \min\{m_0, m_1\}$, for $0 \leq s \leq m_1 \cdot m_2$.

3.2.1.3 Complete Algorithm

Note that Algorithm 1 works properly if both input lists are in decreasing order. Hence, it may be generalised to a number of lists greater than 2 by employing a divide and conquer approach, which works by recursively breaking down the problem into two or more subproblems of the same or related type, until these become simple enough to be solved

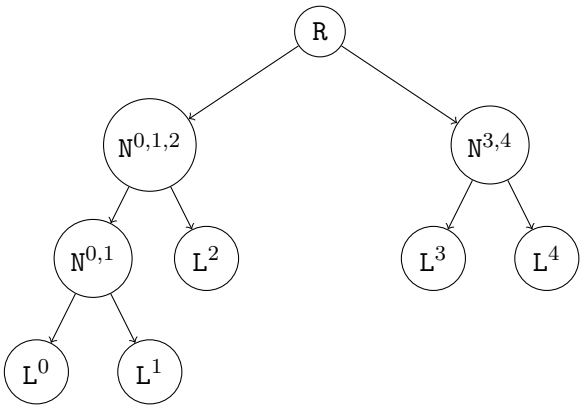


Figure 3.1: Binary tree built from L^0, L^1, L^2, L^3, L^4.

directly. The solutions to the sub-problems are then combined to give a solution to the original problem [17].

To explain the complete algorithm, let us consider the case when there are five chunks as an example. So we have access to five lists of chunk candidates L^i, $0 \leq i < 5$, each of which has a size of m_i. We first call `initialise`$(0, 4)$, as defined in Algorithm 2. This function will build a tree-like structure from the five given lists (see Figure 3.1).

Algorithm 2 creates and initialises each node.

function INITIALISE(i, f)
  if $f = i$ **then**
    $\mathtt{L}^i \leftarrow (\mathtt{null}, \mathtt{null}, \mathtt{null}, \mathtt{null}, \mathtt{null}, L^i)$;
    return $\mathtt{L}^i$
  else
    $q \leftarrow \lfloor \frac{i+f}{2} \rfloor$;
    $\mathtt{N}^{i,\ldots,q} \leftarrow$ `initialise`(i, q);
    $\mathtt{N}^{q+1,\ldots,f} \leftarrow$ `initialise`$(q+1, f)$;
    $\mathtt{c}_0^{i,\ldots,q} \leftarrow$ `getCandidate`$(\mathtt{N}^{i,\ldots,q}, 0)$;
    $\mathtt{c}_0^{q+1,\ldots,f} \leftarrow$ `getCandidate`$(\mathtt{N}^{q+1,\ldots,f}, 0)$;
    $\mathtt{Q}^{i,\ldots,f}$.`add`$(\mathtt{c}_0^{i,\ldots,q}, \mathtt{c}_0^{q+1,\ldots,f}, 0, 0)$;
    $\mathtt{X}_0^{i,\ldots,f} \leftarrow 1; \mathtt{Y}_0^{i,\ldots,f} \leftarrow 1$;
    $\mathtt{N}^{i,\ldots,f} \leftarrow (\mathtt{N}^{i,\ldots,q}, \mathtt{N}^{q+1,\ldots,f}, \mathtt{Q}^{i,\ldots,f}, \mathtt{X}^{i,\ldots,f}, \mathtt{Y}^{i,\ldots,f}, L^{i,\ldots,f})$
    return $\mathtt{N}^{i,\ldots,f}$
  end if
end function

Each node $\mathtt{N}^{i,\ldots,f}$ is a 6-tuple of the form $(\mathtt{N}^{i,\ldots,q}, \mathtt{N}^{q+1,\ldots,f}, \mathtt{Q}^{i,\ldots,f}, \mathtt{X}^{i,\ldots,f}, \mathtt{Y}^{i,\ldots,f}, L^{i,\ldots,f})$, where $\mathtt{N}^{i,\ldots,q}$ and $\mathtt{N}^{q+1,\ldots,f}$ are the children nodes, $\mathtt{Q}^{i,\ldots,f}$ is a priority queue, $\mathtt{X}^{i,\ldots,f}$ and $\mathtt{Y}^{i,\ldots,f}$ are bit vectors and $L^{i,\ldots,f}$ a list of chunk candidates. Additionally, this data structure supports the method `size`(), which returns the maximum number of chunk candidates that this node can generate. This method is easily defined in a recursive way: if $\mathtt{N}^{i,\ldots,f}$ is a leaf node, then the method will return $L^{i,\ldots,f}$.`size`(). Or else, the method will return

$\mathtt{N}^{i,\ldots,q}.\mathtt{size}() \times \mathtt{N}^{q+1,\ldots,f}.\mathtt{size}()$. To avoid computing this value each time this method is called, a node will internally store the value once it has been computed for the first time. Hence, the method will only return the stored value from the second call onwards. Furthermore, the function $\mathtt{getCandidate}(\mathtt{N}^{i,\ldots,f},j)$, as defined in Algorithm 3, returns the j-th best chunk candidate (chunk candidate whose score rank is j) from the node $\mathtt{N}^{i,\ldots,f}$.

Algorithm 3 outputs the j-th best chunk candidate from the node $\mathtt{N}^{i,\ldots,f}$.

```
1: function GETCANDIDATE(N^{i,...,f}, j)
2:     if N^{i,...,f} is a leaf then
3:         return L^{i,...,f}.get(j)
4:     end if
5:     if j ≥ L^{i,...,f}.size() then
6:         L^{i,...,f}.add(nextCandidate(N^{i,...,f}))
7:     end if
8:     return L^{i,...,f}.get(j)
9: end function
```

In order to generate the first N best key candidates from the root node $\mathtt{R}$, with $\mathtt{R} := \mathtt{N}^{0,\ldots,5}$, we simply run $\mathtt{nextCandidate}(\mathtt{R})$, as defined in Algorithm 4, N times. This function internally calls the function `getCandidate` with suitable parameters each time it is required. Calling $\mathtt{getCandidate}(\mathtt{N}^{i,\ldots,f},j)$ may cause this function to internally invoke $\mathtt{nextCandidate}(\mathtt{N}^{i,\ldots,f})$ to generate ordered key candidates from the inner node $\mathtt{N}^{i,\ldots,f}$ on the fly. So any inner node $\mathtt{N}^{i,\ldots,f}$ should keep track of the chunk candidates returned by $\mathtt{getCandidate}(\mathtt{N}^{i,\ldots,f},j)$ when called by its parent, otherwise the j best chunk candidates from $\mathtt{N}^{i,\ldots,f}$ would have to be generated each time such a call is done, which is inefficient.[2] To keep track of the returned chunk candidates, each node $\mathtt{N}^{i,\ldots,f}$ updates its internal list $L^{i,\ldots,f}$ (see lines 5 to 7 in Algorithm 3).

[2] This algorithm is inherently serial.

Algorithm 4 outputs the next highest-scoring chunk candidate from the node $\mathtt{N}^{i,\ldots,f}$.

1: **function** NEXTCANDIDATE($\mathtt{N}^{i,\ldots,f}$)
2: $(\mathtt{c}^x_{j_x}, \mathtt{c}^y_{j_y}, j_x, j_y) \leftarrow \mathtt{Q}^{i,\ldots,f}.\mathtt{poll}()$; $(x = \{i,\ldots,q\}, y = \{q+1,\ldots,f\})$.
3: $\mathtt{X}^{i,\ldots,f}_{j_x} \leftarrow 0; \mathtt{Y}^{i,\ldots,f}_{j_y} \leftarrow 0$;
4: **if** $(j_x + 1) < \mathtt{N}^{i,\ldots,q}.\mathtt{size}()$ **and** $\mathtt{X}^{i,\ldots,f}_{j_x+1}$ is set to 0 **then**
5: $\mathtt{c}^x_{j_x+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{i,\ldots,q}, j_x + 1)$;
6: $\mathtt{Q}^{i,\ldots,f}.\mathtt{add}((\mathtt{c}^x_{j_x+1}, \mathtt{c}^y_{j_y}, j_x + 1, j_y))$;
7: $\mathtt{X}^{i,\ldots,f}_{j_x+1} \leftarrow 1; \mathtt{Y}^{i,\ldots,f}_{j_y} \leftarrow 1$;
8: **end if**
9: **if** $(j_y + 1) < \mathtt{N}^{q+1,\ldots,f}.\mathtt{size}()$ **and** $\mathtt{Y}^{i,\ldots,f}_{j_y+1}$ is set to 0 **then**
10: $\mathtt{c}^y_{j_y+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{q+1,\ldots,f}, j_y + 1)$;
11: $\mathtt{Q}^{i,\ldots,f}.\mathtt{add}((\mathtt{c}^x_{j_x}, \mathtt{c}^y_{j_y+1}, j_x, j_y + 1))$;
12: $\mathtt{X}^{i,\ldots,f}_{j_x} \leftarrow 1; \mathtt{Y}^{i,\ldots,f}_{j_y+1} \leftarrow 1$;
13: **end if**
14: **return** $\mathtt{combine}(\mathtt{c}^x_{j_x}, \mathtt{c}^y_{j_y})$; (the next key candidate).
15: **end function**

3.2.1.4 Memory Consumption

Let us suppose that the encoding of a secret key is $W = 2^{a+b}$ bits in size, and we set $w = 2^a$, and so $\mathcal{N} = 2^b$. Hence, we have access to $\mathcal{N}$ lists L^i, $0 \leq i < 2^b$, each of which has m_i chunk candidates.

Suppose we would like to generate the first N best key candidates. So we first invoke $\mathtt{initialise}(0, \mathcal{N} - 1)$ (Algorithm 2). This call will create a tree-like structure with $b + 1$ levels starting at 0.

- The root node $\mathtt{R} := \mathtt{N}^{0,\ldots,2^b-1}$ at level 0.
- The inner nodes $\mathtt{N}^{I_d} := \mathtt{N}^{i_d}_{\lambda}$ with $I_d = \{i_d \cdot 2^{b-\lambda}, (i_d+1) \cdot 2^{b-\lambda} - 1\}$, where $\lambda, 0 < \lambda < b$, is the level and i_d, $0 \leq i_d < 2^{\lambda}$, is the node identification at the level λ.
- The leaf nodes $\mathtt{L}^i$ at level b, for $0 \leq i < 2^b$.

So this tree will have $2^0 + 2^1 + \cdots + 2^b = 2^{b+1} - 1$ nodes.

Let M_k be the number of bits consumed by chunk candidates stored in memory after calling the function `nextCandidate` with R as a parameter k times. A chunk candidate at level $0 \leq \lambda \leq b$ is of the form $(score, [e_0, \ldots, e_{2^{b-\lambda}-1}])$ with *score* being a real number and e_l being bitstrings. Let B_λ be the number of bits a chunk candidate at level λ occupies in memory.

First note that invoking $\mathtt{initialise}(0, \mathcal{N} - 1)$ causes each internal node's list to grow,

since:

1. At creation of nodes $\mathtt{L}^i$ (lines 2 to 4), $\mathtt{L}^i$ is created by setting $\mathtt{L}^i$'s internal list to L^i and setting $\mathtt{L}^i$'s other components to `null`.
2. At creation of both `R` and nodes $\mathtt{N}_\lambda^{i_d}$, for $0 < \lambda < b-1$ and $0 \leq i_d < 2^\lambda$, the execution of the function `getCandidate` (lines 9 to 10) makes their corresponding left child (right child) store a new chunk candidate in their corresponding internal list. This is, for $0 < \lambda \leq b-1$, $0 \leq i_d < 2^\lambda$, the $\mathtt{N}_\lambda^{i_d}$'s internal list has a new element.

Therefore, $M_0 = \sum_{\lambda=1}^{b-1} 2^\lambda B_\lambda + B_b(\sum_{i=0}^{2^b-1} m_i)$.

Suppose the best key candidate is about to be generated, then `nextCandidate(R)` will be executed for the first time. This routine will remove the extended candidate $(\mathtt{c}_0^x, \mathtt{c}_0^y, 0, 0)$ out of `R`'s priority queue. If it enters the first **if** (lines 4 to 8), it will make the call `getCandidate`$(\mathtt{N}_1^0, 1)$ (line 5), which may cause each node, except for the leaf nodes, of the left subtree to store at most a new chunk candidate in its corresponding internal list. Hence, retrieving the chunk candidate $\mathtt{c}_1^x$ may cause at most $2^{\lambda-1}$ chunk candidates per level λ, $1 \leq \lambda < b$, to be stored. Likewise, if it enters the second **if** (lines 9 to 13), it will call the function `getCandidate`$(\mathtt{N}_1^1, 1)$ (line 10), which may cause each node, except for the leaf nodes, of the right subtree to store at most a new chunk candidate in its corresponding internal list. Therefore, retrieving the chunk candidate $\mathtt{c}_1^y$ (line 10) may cause at most $2^{\lambda-1}$ chunk candidates per level λ, $1 \leq \lambda < b$, to be stored. Therefore, after generating the best key candidate, $p_\lambda^{(1)} \leq 2^\lambda$ chunk candidates per level λ, $1 \leq \lambda < b$, will be stored in memory, hence $M_0 \leq M_1 = M_0 + \sum_{\lambda=1}^{b-1} p_\lambda^{(1)} B_\lambda \leq 2\sum_{\lambda=1}^{b-1} 2^\lambda B_\lambda + B_b(\sum_{i=0}^{2^b-1} m_i)$ bits are consumed by chunk candidates stored in memory.

Let us assume that $k-1$ key candidates have already been generated, therefore M_{k-1} bits are consumed by chunk candidates in memory, with $M_{k-1} = M_0 + \sum_{d=1}^{k-1}\sum_{\lambda=1}^{b-1} p_\lambda^{(d)} B_\lambda$.

Let us suppose the k-th best key candidate is about to be generated, then the method `nextCandidate(R)` will be executed for the k-th time. This routine will remove the best extended candidate $(\mathtt{c}_{j_x}^x, \mathtt{c}_{j_y}^y, j_x, j_y)$ out of the `R`'s priority queue. It will then attempt to insert two new extended candidates into `R`'s priority queue. As seen previously, retrieving the chunk candidate $\mathtt{c}_{j_x+1}^x$ may cause at most $2^{\lambda-1}$ chunk candidates per level λ, $1 \leq \lambda < b$, to be stored. Likewise, retrieving the chunk candidate $\mathtt{c}_{j_y+1}^y$ may also cause at most $2^{\lambda-1}$

chunk candidates per level λ, $1 \leq \lambda < b$, to be stored. Therefore, after generating the k-th best key candidate, $p_\lambda^{(k)} \leq 2^\lambda$ chunk candidates per level λ, $1 \leq \lambda < b$, will be stored in memory, hence $M_k = M_{k-1} + \sum_{\lambda=1}^{b-1} p_\lambda^{(k)} B_\lambda = M_0 + \sum_{d=1}^{k} \sum_{\lambda=1}^{b-1} p_\lambda^{(d)} B_\lambda$ bits are consumed by chunk candidates stored in memory.

It follows that if N candidate keys are generated, then

$$M_N = M_0 + \sum_{d=1}^{N} \sum_{\lambda=1}^{b-1} p_\lambda^{(d)} B_\lambda = \sum_{\lambda=1}^{b-1} 2^\lambda B_\lambda + B_b(\sum_{i=0}^{2^b-1} m_i) + \sum_{d=1}^{N} \sum_{\lambda=1}^{b-1} p_\lambda^{(d)} B_\lambda,$$

bits are consumed by chunk candidates stored in memory in addition to the extended candidates stored internally in the priority queue of the nodes R and $\mathtt{N}_\lambda^{i_d}$. Therefore, this algorithm may consume a large amount of memory if it is used to generate a large number of key candidates, which may be problematic.

3.2.2 A Bounded-Space Near-Optimal Key Enumeration Algorithm

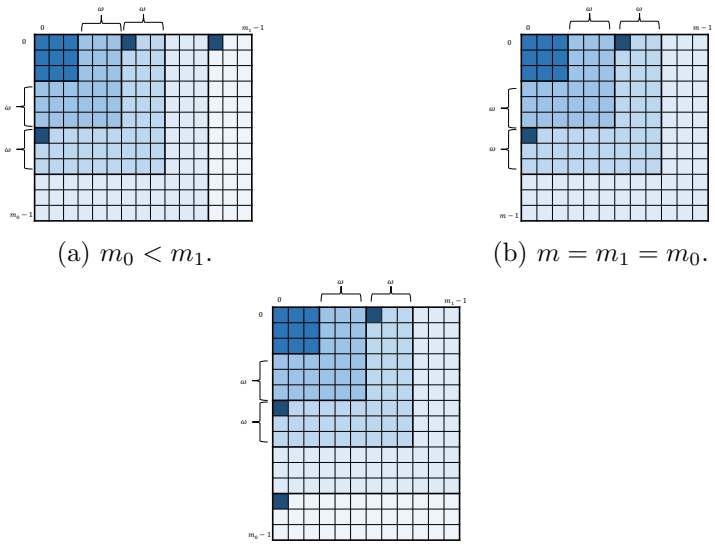
We next will describe a key enumeration algorithm introduced in [18]. This algorithm builds upon OKEA and can enumerate a large number of key candidates without exceeding the available space. The trade-off is that the enumeration order is only near-optimal, rather than optimal as it is in OKEA. We firstly will give the basic idea behind the algorithm by assuming the encoding of the secret key is represented as two chunks, hence we have access to two lists of chunk candidates.

3.2.2.1 Basic Algorithm

Let $L^0 = [\mathsf{c}_0^0, \mathsf{c}_2^0, \ldots, \mathsf{c}_{m_0-1}^0]$ and $L^1 = [\mathsf{c}_0^1, \mathsf{c}_2^1, \ldots, \mathsf{c}_{m_1-1}^1]$ be the two lists respectively and let $\omega > 0$ be an integer such that $\omega \mid m_0$ and $\omega \mid m_1$. Each list is in decreasing order based on the score component of its chunk candidates. Let us set $m_{min} = \min\{m_0, m_1\}$, and define R_{k_0,k_1} as $R_{k_0,k_1} = \{0, \ldots, k_0 \cdot \omega - 1\} \times \{0, \ldots, k_1 \cdot \omega - 1\}$, where k_0, k_1 are positive integers. The key space is divided into layers $layer_k^\omega$ of width ω. Figure 3.2 depicts each layer as a different shade of the blue colour. Formally,

$$layer_k^\omega := \{(\mathsf{c}_{j_0}^0, \mathsf{c}_{j_1}^1) \mid (j_0, j_1) \in R_{k,k} \setminus R_{k-1,k-1}\},$$

for $1 \leq k \leq \frac{m_{min}}{\omega}$. The remaining layers are defined as:



(a) $m_0 < m_1$. (b) $m = m_1 = m_0$.

(c) $m_0 > m_1$.

Figure 3.2: Geometric representation of the key space divided into layers of width $\omega = 3$.

If $m_0 \geq m_1$,

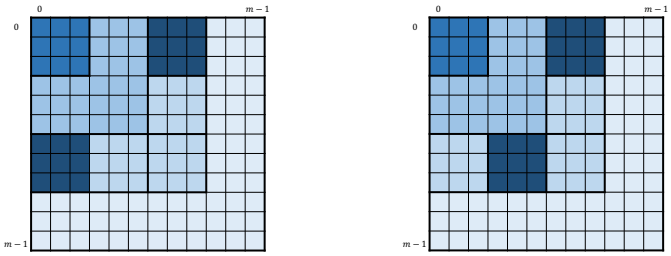
$$layer_k^\omega := \{(\mathsf{c}_{j_0}^0, \mathsf{c}_{j_1}^1) \mid (j_0, j_1) \in R_{k,\frac{m_{min}}{\omega}} \setminus R_{\frac{m_{min}}{\omega},\frac{m_{min}}{\omega}}\},$$

for $\frac{m_{min}}{\omega} < k \leq \frac{m_0}{\omega}$. Or else,

$$layer_k^\omega := \{(\mathsf{c}_{j_0}^0, \mathsf{c}_{j_1}^1) \mid (j_0, j_1) \in R_{\frac{m_{min}}{\omega},k} \setminus R_{\frac{m_{min}}{\omega},\frac{m_{min}}{\omega}}\},$$

for $\frac{m_{min}}{\omega} < k \leq \frac{m_1}{\omega}$.

The ω-layer key enumeration algorithm: Divide the key-space into layers of width ω. Then, go over $layer_k^\omega$, one by one, in increasing order. For each $layer_k^\omega$, enumerate its key candidates by running OKEA within the layer $layer_k^\omega$. More specifically, for each $layer_k^\omega$, $1 \leq k \leq \frac{m_{min}}{\omega}$, the algorithm inserts the two corners, i.e. the extended candidates $(\mathsf{c}_{(k-1)\cdot\omega}^0, \mathsf{c}_0^1, (k-1)\cdot\omega, 0), (\mathsf{c}_0^0, \mathsf{c}_{(k-1)\cdot\omega}^1, 0, (k-1)\cdot\omega)$, into the data structure `Q`. The algorithm then proceeds to extract extended candidates and insert their successors as usual, but limiting the algorithm not to exceed the boundaries of the layer $layer_k^\omega$ when selecting components of candidates. For the remaining layers, if any, the algorithm inserts only one corner, either the extended candidate $(\mathsf{c}_{(k-1)\cdot\omega}^0, \mathsf{c}_0^1, (k-1)\cdot\omega, 0)$ or the extended candidate $(\mathsf{c}_0^0, \mathsf{c}_{(k-1)\cdot\omega}^1, 0, (k-1)\cdot\omega)$, into the data structure `Q` and then proceeds as usual, while not exceeding the boundaries of the layer. Figure 3.2 also shows the extended candidates (represented as the smallest squares in a strong shade of blue within a layer) to be inserted into `Q` when a certain layer will be enumerated.



(a) Step 1. (b) Step 2.

Figure 3.3: Geometric representation of the key enumeration within $layer_3^3$.

3.2.2.2 Complete Algorithm

When the number of chunks is greater than 2, the algorithm applies a recursive decomposition of the problem (similar to OKEA). Whenever a new chunk candidate is inserted into the candidate set, its value is obtained by applying the enumeration algorithm to the lower level. We explain an example to give an idea of the general algorithm. Let us suppose the encoding of the secret key is divided into 4 chunks, then we have access to 4 lists of chunk candidates, each of which is of size m_i with $\omega \mid m_i$.

To generate key candidates, we need to generate the two lists of chunk candidates for the lower level $L^{0,1}$ and $L^{2,3}$ on the fly as far as required. For this, we maintain a set of next potential candidates, for each dimension – $\mathtt{Q}^{0,1}$ and $\mathtt{Q}^{2,3}$, so that each next chunk candidate obtained from $\mathtt{Q}^{0,1}$ (or $\mathtt{Q}^{2,3}$) is stored in the list $L^{0,1}$ (or $L^{2,3}$). Because the enumeration is performed by layers, the sizes of the data structures $\mathtt{Q}^{1,2}$ and $\mathtt{Q}^{3,4}$ are bounded by 2ω. However, this is not the case for the lists $L^{0,1}$ and $L^{2,3}$, which grow as the number of candidates enumerated grows, hence becoming problematic as seen in Section 3.2.1.4.

To handle this, each $layer_k^\omega$ is partitioned into squares of size $\omega \times \omega$. The algorithm still enumerates the key candidates in $layer_1^\omega$ first, then in $layer_2^\omega$ and so on, but in each $layer_k^\omega$ the enumeration will be square-by-square. Figure 3.3 depicts the geometric representation of the key enumeration within $layer_3^3$, where a square (strong shade of blue) within a layer represents the square being processed by the enumeration algorithm. More specifically, for given non-negative integers I, J, let us define $S_{I,J}^\omega$ as

$$S_{I,J}^\omega := \{(\mathtt{c}_{j_x}, \mathtt{c}_{j_y}) \mid I \cdot \omega \leq j_x < (I+1) \cdot \omega, J \cdot \omega \leq j_y < (J+1) \cdot \omega\}.$$

Algorithm 5 creates and initialises each node.

1: **function** INITIALISE(i, f)
2: **if** $f = i$ **then**
3: $\mathtt{L}^i \leftarrow (\mathtt{null}, \mathtt{null}, \mathtt{null}, \mathtt{null}, \mathtt{null}, \mathtt{null}, \mathtt{null}, L^i)$;
4: **return** $\mathtt{L}^i$
5: **else**
6: $q \leftarrow \lfloor \frac{i+f}{2} \rfloor$;
7: $\mathtt{N}_0^{i,\ldots,q} \leftarrow \mathtt{initialise}(i, q)$;
8: $\mathtt{N}_0^{q+1,\ldots,f} \leftarrow \mathtt{initialise}(q+1, f)$;
9: $\mathtt{N}_1^{i,\ldots,q} \leftarrow \mathtt{initialise}(i, q)$;
10: $\mathtt{N}_1^{q+1,\ldots,f} \leftarrow \mathtt{initialise}(q+1, f)$;
11: $\mathtt{c}_0^{i,\ldots,q} \leftarrow \mathtt{getCandidate}(\mathtt{N}_0^{i,\ldots,q}, 0, 2)$;
12: $\mathtt{c}_0^{q+1,\ldots,f} \leftarrow \mathtt{getCandidate}(\mathtt{N}_1^{q+1,\ldots,f}, 0, 2)$;
13: $\mathtt{Q}^{i,\ldots,f}.\mathtt{add}(\mathtt{c}_0^{i,\ldots,q}, \mathtt{c}_0^{q+1,\ldots,f}, 0, 0)$;
14: $\mathtt{X}_0^{i,\ldots,f} \leftarrow 1; \mathtt{Y}_0^{i,\ldots,f} \leftarrow 1$;
15: $\mathtt{N}^{i,\ldots,f} \leftarrow (\mathtt{N}_0^{i,\ldots,q}, \mathtt{N}_0^{q+1,\ldots,f}, \mathtt{N}_1^{i,\ldots,q}, \mathtt{N}_1^{q+1,\ldots,f}, \mathtt{Q}^{i,\ldots,f}, \mathtt{X}^{i,\ldots,f}, \mathtt{Y}^{i,\ldots,f}, L^{i,\ldots,f})$
16: **return** $\mathtt{N}^{i,\ldots,f}$
17: **end if**
18: **end function**

Let us set $m_{min} = \min\{m_0 \cdot m_1, m_2 \cdot m_3\}$, hence

$$layer_k^\omega = S_{k-1,0}^\omega \cup S_{k-1,1}^\omega \cup \ldots \cup S_{k-1,k-1}^\omega \cup S_{k-2,k-1}^\omega \cup \ldots \cup S_{0,k-1}^\omega,$$

for $1 \leq k \leq \frac{m_{min}}{\omega}$. The remaining layers, if any, are also partitioned in a similar way.

The in-layer algorithm then proceeds as follows. For each $layer_k^\omega$, $1 \leq k \leq \frac{m_{min}}{\omega}$, the in-layer algorithm first enumerates the candidates in the two corner squares $S = S_{k-1,0} \cup S_{0,k-1}$ by applying OKEA on S. At some point, one of the two squares is completely enumerated. Assume this is $S_{k-1,0}$. At this point, the only square that contains the next key candidates after $S_{k-1,0}$ is the successor $S_{k-1,1}$. Therefore, when one of the squares is completely enumerated, its successor is inserted in S, as long as S does not contain a square in the same row or column. For the remaining layers, if any, the in-layer algorithm first enumerates the candidates in the square $S = S_{k-1,0}$ (or $S_{0,k-1}$) by applying OKEA on it. Once the square is completely enumerated, its successor is inserted in S, and so on. This in-layer partition into squares reduces the space complexity, since instead of storing the full list of chunk candidates of the lower levels, only the relevant chunk candidates are stored for enumerating the two current squares.

Because this in-layer algorithm enumerates at most two squares at any time in a layer, the tree-like structure is no longer a binary tree. A node $\mathtt{N}^{i,\ldots,f}$ is now extended to an 8-tuple of the form $(\mathtt{N}_0^{i,\ldots,q}, \mathtt{N}_0^{q+1,\ldots,f}, \mathtt{N}_1^{i,\ldots,q}, \mathtt{N}_1^{q+1,\ldots,f}, \mathtt{Q}^{i,\ldots,f}, \mathtt{X}^{i,\ldots,f}, \mathtt{Y}^{i,\ldots,f}, L^{i,\ldots,f})$, where $\mathtt{N}_b^{i,\ldots,q}$ and $\mathtt{N}_b^{q+1,\ldots,f}$, for $b = 0, 1$, are the children nodes used to enumerate at most two squares

Algorithm 6 outputs the j-th chunk candidate from the node $\mathtt{N}^{i,\ldots,f}$.

```
1: function GETCANDIDATE(N^{i,...,f}, j, sw)
2:     if N^{i,...,f} is a leaf then
3:         return L^{i,...,f}.get(j)
4:     end if
5:     if sw = 0 then
6:         restart(N^{i,...,f});
7:     else
8:         if sw = 1 then
9:             L^{i,...,f}.clear();
10:        end if
11:    end if
12:    j ← j mod ω
13:    if j ≥ L^{i,...,f}.size() then
14:        L^{i,...,f}.add(nextCandidate(N^{i,...,f}))
15:    end if
16:    return L^{i,...,f}.get(j)
17: end function
```

in a particular layer, $\mathtt{Q}^{i,\ldots,f}$ is a priority queue, $\mathtt{X}^{i,\ldots,f}$ and $\mathtt{Y}^{i,\ldots,f}$ are bit vectors and $L^{i,\ldots,f}$ a list of chunk candidates. Hence, the function that initialises the tree-like structure is adjusted to create the two additional children for a given node (see Algorithm 5).

Moreover, the function `getCandidate`($\mathtt{N}^{i,\ldots,f}, j, sw$) is also adjusted so that each node's internal list $L^{i,\ldots,f}$ has at most ω chunk candidates at any stage of the algorithm (see Algorithm 6). This function internally makes the call `restart`($\mathtt{N}^{i,\ldots,f}$) if $sw = 0$. The call `restart`($\mathtt{N}^{i,\ldots,f}$) causes $\mathtt{N}^{i,\ldots,f}$ to restart its enumeration, i.e., after `restart`($\mathtt{N}^{i,\ldots,f}$) has been invoked, calling `nextCandidate`($\mathtt{N}^{i,\ldots,f}$) will return the first chunk candidate from $\mathtt{N}^{i,\ldots,f}$. Also, the function `getHighestScoreCandidate`($S^{\omega}_{I,J}$) returns the highest-scoring extended candidate from the square $S^{\omega}_{I,J}$. Note this function is called to get the highest-scoring extended candidate from the successor of $S_{I,J}$.[3] Finally, Algorithm 7 precisely describes the manner in which this enumeration works.

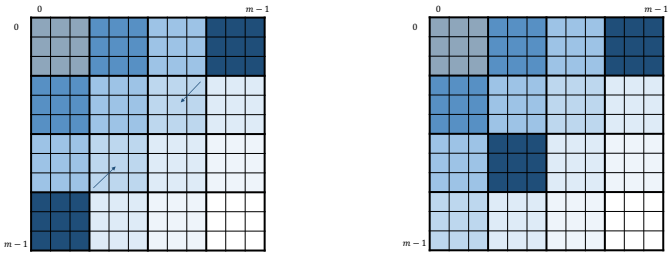
3.2.2.3 Parallelisation

The original authors of [18] suggest having OKEA run in parallel per square within a layer, but this has a negative effect on the algorithm's near-optimality property and even on its overall performance since there are squares within a layer that are strongly dependent on others, i.e., for the algorithm to enumerate the successor square, say, $S_{I,J+1}$ within a layer, it requires to have information that is obtained during the enumeration of $S_{I,J}$. Hence, this strategy may incur extra computation and also be difficult to implement.

[3] At this point, the content of the internal list of $\mathtt{N}_0^{q+1,\ldots,f}$ is cleared, if $b = 0$. Otherwise, the content of the internal list of $\mathtt{N}_1^{i,\ldots,q}$ is cleared, if $b = 1$.

Algorithm 7 outputs the next chunk candidate from the node $\mathtt{N}^{i,\dots,f}$.

1: **function** NEXTCANDIDATE($\mathtt{N}^{i,\dots,f}$).
2: $(\mathtt{c}^x_{j_x}, \mathtt{c}^y_{j_y}, j_x, j_y) \leftarrow \mathtt{Q}^{i,\dots,f}.\mathtt{poll}()$; $(x = \{i,\dots,q\}, y = \{q+1,\dots,f\})$.
3: $\mathtt{X}^{i,\dots,f}_{j_x} \leftarrow 0; \mathtt{Y}^{i,\dots,f}_{j_y} \leftarrow 0$;
4: $I \leftarrow \lfloor \frac{j_x}{\omega} \rfloor; J \leftarrow \lfloor \frac{j_y}{\omega} \rfloor; b = (I \geq J)\ ?\ 0 : 1$;
5: **if** $S_{I,J}$ is completely enumerated **then**
6: $last_I \leftarrow \mathtt{N}^{i,\dots,q}_0.\mathtt{size}()/\omega - 1$;
7: $last_J \leftarrow \mathtt{N}^{q+1,\dots,f}_1.\mathtt{size}()/\omega - 1$;
8: **if** $I = J$ **or** $(I > last_J$ **and** $J = last_J)$ **or** $(J > last_I$ **and** $I = last_I)$ **then**
9: **if** $(j_x + 1) < (last_I + 1) \cdot \omega$ **then**
10: $\mathtt{c}^x_{j_x+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{i,\dots,q}_0, j_x + 1, 1)$;
11: $\mathtt{c}^y_0 \leftarrow \mathtt{getCandidate}(\mathtt{N}^{q+1,\dots,f}_0, 0, 0)$;
12: $\mathtt{Q}^{i,\dots,f}.\mathtt{add}((\mathtt{c}^x_{j_x+1}, \mathtt{c}^y_0, j_x + 1, 0))$;
13: $\mathtt{X}^{i,\dots,f}_{j_x+1} \leftarrow 1; \mathtt{Y}^{i,\dots,f}_0 \leftarrow 1$;
14: **end if**
15: **if** $(j_y + 1) < (last_J + 1) \cdot \omega$ **then**
16: $\mathtt{c}^x_0 \leftarrow \mathtt{getCandidate}(\mathtt{N}^{i,\dots,q}_1, 0, 0)$;
17: $\mathtt{c}^y_{j_y+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{q+1,\dots,f}_1, j_y + 1, 1)$;
18: $\mathtt{Q}^{i,\dots,f}.\mathtt{add}((\mathtt{c}^x_0, \mathtt{c}^y_{j_y+1}, 0, j_y + 1))$;
19: $\mathtt{X}^{i,\dots,f}_0 \leftarrow 1; \mathtt{Y}^{i,\dots,f}_{j_y+1} \leftarrow 1$;
20: **end if**
21: **else**
22: **if** no candidates in same row/column as $\mathtt{Successor}(S_{I,J})$ **then**
23: $(\mathtt{c}^x_k, \mathtt{c}^y_l, k, l) \leftarrow \mathtt{getHighestScoreCandidate}(\mathtt{Successor}(S_{I,J}))$;
24: $\mathtt{Q}^{i,\dots,f}.\mathtt{add}((\mathtt{c}^x_k, \mathtt{c}^y_l, k, l))$;
25: $\mathtt{X}^{i,\dots,f}_k \leftarrow 1; \mathtt{Y}^{i,\dots,f}_l \leftarrow 1$;
26: **end if**
27: **end if**
28: **else**
29: **if** $(j_x + 1, j_y) \in S_{I,J}$ **and** $\mathtt{X}^{i,\dots,f}_{j_x+1}$ is set to 0 **then**
30: $\mathtt{c}^x_{j_x+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{i,\dots,q}_b, j_x + 1, 2)$;
31: $\mathtt{Q}^{i,\dots,f}.\mathtt{add}((\mathtt{c}^x_{j_x+1}, \mathtt{c}^y_{j_y}, j_x + 1, j_y))$;
32: $\mathtt{X}^{i,\dots,f}_{j_x+1} \leftarrow 1; \mathtt{Y}^{i,\dots,f}_{j_y} \leftarrow 1$;
33: **end if**
34: **if** $(j_x, j_y + 1) \in S_{I,J}$ **and** $\mathtt{Y}^{i,\dots,f}_{j_y+1}$ is set to 0 **then**
35: **if** $I = J$ **then**
36: $\mathtt{c}^y_{j_y+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{q+1,\dots,f}_1, j_y + 1, 2)$;
37: **else**
38: $\mathtt{c}^y_{j_y+1} \leftarrow \mathtt{getCandidate}(\mathtt{N}^{q+1,\dots,f}_b, j_y + 1, 2)$;
39: **end if**
40: $\mathtt{Q}^{i,\dots,f}.\mathtt{add}((\mathtt{c}^x_{j_x}, \mathtt{c}^y_{j_y+1}, j_x, j_y + 1))$;
41: $\mathtt{X}^{i,\dots,f}_{j_x} \leftarrow 1; \mathtt{Y}^{i,\dots,f}_{j_y+1} \leftarrow 1$;
42: **end if**
43: **end if**
44: **return** $\mathtt{combine}(\mathtt{c}^x_{j_x}, \mathtt{c}^y_{j_y})$;
45: **end function**



(a) Step 1. (b) Step 2.

Figure 3.4: Geometric representation of the key enumeration by variant.

3.2.2.4 Variant

As a variant of this algorithm, we propose to slightly change the definition of layer. Here a layer consists of all the squares within a secondary diagonal, as shown in Figure 3.4. The variant will follow the same process as the original algorithm, i.e. enumeration layer by layer starting at the first secondary diagonal. Within each layer, it will first enumerate the two square corners $S = S_{k-1,0} \cup S_{0,k-1}$, by applying OKEA on it. Once one of two squares is enumerated, let us say $S_{k-1,0}$, its successor $S_{k-2,1}$ will be inserted in S as long as such insertion is possible. The algorithm will continue the enumeration by applying OKEA on the updated S and so on. This algorithm is motivated by the intuition that enumerating secondary diagonals may improve the quality of order of output key candidates, i.e., it may be nearer to optimal. This variant, however, may have a potential disadvantage in the multidimensional case, because it strongly depends on having all the previously enumerated chunk candidates of both dimension x and y stored. To illustrate this, let us suppose that this square $S_{k-2,1}$ is to be inserted. Then the algorithm needs to insert its highest scoring extended candidate, $(\mathsf{c}^x_{(k-2)\cdot\omega}, \mathsf{c}^y_{\omega}, (k-2)\cdot\omega, \omega)$, into the queue. Hence, the algorithm needs to somehow have both $\mathsf{c}^x_{(k-2)\cdot\omega}$ and c^y_{ω} readily accessible when needed. This implies the need to store them when they are being enumerated (in previous layers). Comparatively, the original algorithm only requires having the w previously generated chunk candidates of both dimension x and y stored, which is advantageous in terms of memory consumption.

3.2.3 A Simple Stack-Based, Depth-First Key Enumeration Algorithm

We next present a memory-efficient, non-optimal key enumeration algorithm that generates key candidates whose total scores are within a given interval $[B_1, B_2]$ that is based on the algorithm introduced by Martin *et al.* in [48]. We note that the original algorithm is

fairly efficient while generating a new key candidate, however its overall performance may be negatively affected by its use of memory, since it was originally designed to store each new generated key candidate, each of which is tested only once the algorithm has completed the enumeration. Our variant, however, makes use of a stack (LIFO queue) [17] during the enumeration process. This helps in maintaining the state of the algorithm. Each newly generated key candidate may be tested immediately and there is no need for candidates to be stored for future processing.

Our variant basically performs a depth-first search in an undirected graph G originated from the $\mathcal{N}$ lists of chunk candidates $L^i = [\mathsf{c}_0^i, \mathsf{c}_2^i, \ldots, \mathsf{c}_{m_i-1}^i]$. This graph G has $\sum_{i=0}^{\mathcal{N}-1} m_i$ vertices, each of which represents a chunk candidate. Each vertex v_j^i is connected to the vertices v_k^{i+1}, $0 \leq i < \mathcal{N}-1$, $0 \leq j < m_i, 0 \leq k < m_{i+1}$. At any vertex v_j^i, the algorithm will check if c_j^i.score plus an accumulated score is within the given interval $[B_1, B_2]$. If so, it will select the chunk candidate c_j^i for the chunk i and travel forward to the vertex v_0^{i+1}. Or else, it will continue exploring and attempt to travel to the vertex v_{j+1}^i. Otherwise, it will travel backwards to a vertex from the previous chunk v_k^{i-1}, $0 \leq k < m_{i-1}$, when there is no suitable chunk candidate for the current chunk i.

3.2.3.1 Setup

We now introduce a couple of tools that we will use to describe the algorithm, using the following notations. `S` will denote a stack. This data structure supports two basic methods [17]. Firstly, the method `S.pop()` removes the element at the top of this stack and returns that element as the value of this function. Secondly, the method `S.push`(e) pushes e onto the top of this stack. This stack `S` will store 4-tuples of the form $(score, i, j, indices)$, where $score$ is the accumulated score at any stage of the algorithm, i, j are the indices for the chunk candidate c_j^i, and $indices$ is an array of positive integers holding the indices of the selected chunk candidates, i.e., the chunk candidate $\mathsf{c}_{indices[k]}^k$ is assigned to chunk k, for each k, $0 \leq k \leq i$.

Algorithm 8 outputs a key candidate in the interval $[B_1, B_2]$.

```
1: function nextCandidate(S,B1,B2).
2:     while S is not empty do
3:         (aScore, i, j, indices) ← S.pop();
4:         if j < L^i.size() − 1 then
5:             S.push((aScore, i, j + 1, indices));
6:         end if
7:         uScore ← aScore + c^i_j.score;
8:         if uScore ≤ B2 then
9:             if i = N − 1 then
10:                if B1 ≤ uScore then
11:                    indices ← s.indices || [j];.
12:                    c ← combine(c^0_{indices[0]}, ..., c^{N−1}_{indices[N−1]});
13:                    break;
14:                end if
15:            else
16:                S.push((uScore, i + 1, 0, indices || [j]));
17:            end if
18:        end if
19:    end while
20:    return c
21: end function
```

3.2.3.2 Complete Algorithm

Firstly, at the initialisation stage, the 4-tuple $(0, 0, 0, [])$ will be inserted into the stack `S`. The main loop of this algorithm will call the function `nextCandidate`(`S`,B_1,B_2), defined in Algorithm 8, as long as the stack `S` is not empty. Specifically the main loop will call this function to obtain a key candidate whose score is in the range $[B_1, B_2]$. Algorithm 8 will then attempt to find such a candidate and once it has found such a candidate, it will return the candidate to the main loop (at this point `S` may not be empty). The main loop will get the key candidate, process or test it and continue calling the function `nextCandidate`(`S`,B_1,B_2) as long as `S` is not empty. Because of the use of the stack `S`, the state of Algorithm 8 will not be lost, therefore each time the main loop calls it, it will return a new key candidate whose score lie in the interval $[B_1, B_2]$. The main loop will terminate once all possible key candidates whose scores are within the interval $[B_1, B_2]$ have already been generated, which will happen once the stack is empty.

3.2.3.3 Speeding up the Pruning Process

We here propose a small improvement to speed up the pruning process. This makes use of two precomputed tables `minArray`[i](`maxArray`[i]). The entry `minArray`[i] (`maxArray`[i]) holds the global minimum (maximum) value that can be reached from chunk i to chunk

$\mathcal{N}-1$. In other words,

$$\texttt{minArray}[i] = \min\ \{\sum_{j=i}^{\mathcal{N}-1} \mathtt{c}_{k_j}^j.score\ :\ \mathtt{c}_{k_j}^j \in L^j\},\ 0 \le i < \mathcal{N},$$

$$\texttt{maxArray}[i] = \max\ \{\sum_{j=i}^{\mathcal{N}-1} \mathtt{c}_{k_j}^j.score\ :\ \mathtt{c}_{k_j}^j \in L^j\},\ 0 \le i < \mathcal{N},$$

$$\texttt{minArray}[\mathcal{N}] = \texttt{maxArray}[\mathcal{N}] = 0.$$

When each list of chunk candidates $L^i = [\mathtt{c}_0^i, \mathtt{c}_1^i, \ldots, \mathtt{c}_{m_i-1}^i], 0 \le i < \mathcal{N}$ is in decreasing order based on the score component of its chunk candidates, we can compute the entry `minArray`[i] (`maxArray`[i]) simply by computing

$$\texttt{minArray}[i] = \sum_{j=i}^{\mathcal{N}-1} \mathtt{c}_{m_j-1}^j.score,$$

and

$$\texttt{maxArray}[i] = \sum_{j=i}^{\mathcal{N}-1} \mathtt{c}_0^j.score.$$

Algorithm 8 is sped up by computing `maxS` (`minS`), which is the maximum(minimum) score that can be obtained from the current chunk candidate, and then checking if the intersection of the intervals $[\texttt{minS}, \texttt{maxS}]$ and $[B_1, B_2]$ is not empty. Algorithm 9 shows the improved enumeration algorithm.

3.2.3.4 Memory Consumption

We claim that at any stage of the algorithm, there are at most $\mathcal{N}$ 4-tuples stored in the stack `S`. Indeed, after the stack is initialised, it only contains the 4-tuple $(0, 0, 0, [])$. Note that during the execution of a **while** iteration, a 4-tuple is removed out of the stack and two new 4-tuples might be inserted. Hence, after s **while** iterations have been completed, there will be $N_{\mathtt{S}}^s = 1 + (-1 + l_1) + (-1 + l_2) + (-1 + l_3) + (-1 + l_4) + \cdots + (-1 + l_s)$ 4-tuples, where $0 \le l_r \le 2$, for $1 \le r \le s$.

Algorithm 9 outputs a key candidate in the interval $[B_1, B_2]$.

```
1: function nextCandidateI(S,B1,B2).
2:     while S is not empty do
3:         (aScore, i, j, indices) ← S.pop();
4:         if j < L^i.size() − 1 then
5:             S.push((aScore, i, j + 1, indices));
6:         end if
7:         uScore ← aScore + c_j^i.score;
8:         maxS ← uScore + maxArray[i + 1];
9:         minS ← uScore + minArray[i + 1];
10:        if maxS ≥ B1 and minS ≤ B2 then
11:            if uScore ≤ B2 then
12:                if i = N − 1 then
13:                    if B1 ≤ uScore then
14:                        indices ← indices || [j];
15:                        c ← combine(c^0_{indices[0]}, ..., c^{N−1}_{indices[N−1]});
16:                        break;
17:                    end if
18:                else
19:                    S.push((uScore, i + 1, 0, indices || [j]));
20:                end if
21:            end if
22:        end if
23:    end while
24:    return c;
25: end function
```

Suppose now that the algorithm is about to execute the k-th **while** iteration during which the first valid key candidate will be returned. Therefore, $N_{\mathsf{S}}^{k-1} = 1 + (-1 + l_1) + (-1 + l_2) + (-1 + l_3) + (-1 + l_4) + \cdots + (-1 + l_{k-1}) \leq \mathcal{N}$. During the execution of the k-th **while** iteration, a 4-tuple will be removed and only a new 4-tuple will be considered for insertion in the stack. Therefore, we have that $N_{\mathsf{S}}^{k} = N_{\mathsf{S}}^{k-1} - 1 + l_k \leq \mathcal{N} - 1 + l_k \leq \mathcal{N}$, since $0 \leq l_k \leq 1$. Applying a similar reasoning, we have $N_{\mathsf{S}}^{n} \leq \mathcal{N}$ for $n > k$.

3.2.3.5 Parallelisation

One of the most interesting features of the previous algorithm is that it is parallelizable. The original authors suggested as a parallelisation method to run instances of the algorithm over different disjoint intervals [48]. Although this parallelisation method is clearly effective and has a potential advantage as the different instances will produce non-overlapping lists of key candidates with the instance searching over the first interval producing the most-likely key candidates, it is not efficient, since each instance will inevitably repeat a lot of the work done by the other instances. We here propose another parallelisation method that partitions the search space to avoid the repetition of work.

Suppose that we want to have t parallel, independent tasks $T_1, T_2, T_3, \ldots, T_t$ to search over

a given interval in parallel. Let $L^i = [\mathsf{c}_0^i, \mathsf{c}_2^i, \ldots, \mathsf{c}_{m_i-1}^i]$ be the list of chunk candidates for chunk i, $0 \leq i \leq \mathcal{N} - 1$.

We first assume that $t \leq m_0$, where m_0 is the size of L^0. In order to construct these tasks, we partition L^0 into t disjoint, roughly equal-sized sublists L_j^0, $1 \leq j \leq t$. We set each task T_j to perform its enumeration over the given interval but only considering the lists of chunk candidates $L_j^0, L^1, \ldots, L^{\mathcal{N}-1}$.

The previous method can be easily generalised for $m_0 < t \ll \prod_{k=0}^{\mathcal{N}-1} m_k$. Indeed, first find the smallest integer l, with $0 < l < \mathcal{N} - 1$, such that $\prod_{k=0}^{l-1} m_k < t \leq \prod_{k=0}^{l} m_k$. We then construct the list of chunk candidates $L^{0,\ldots,l}$ as follows. For each $(l+1)-$tuple $(\mathsf{c}_{j_0}^0, \mathsf{c}_{j_1}^1, \ldots, \mathsf{c}_{j_l}^l)$, with $\mathsf{c}_{j_k}^k \in L^k, 0 \leq j_k < m_k, 0 \leq k \leq l$, the chunk candidate $\mathsf{c}^{j_0,\ldots,j_l}$ is constructed by calculating $\mathsf{c}^{j_0,\ldots,j_l}.score = \sum_{k=0}^{l} \mathsf{c}_{j_k}^k.score$ and setting $\mathsf{c}^{j_0,\ldots,j_l}.value = [\mathsf{c}_{j_0}^0.value, \ldots, \mathsf{c}_{j_l}^l.value]$, and then $\mathsf{c}^{j_0,\ldots,j_l}$ is added to $L^{0,\ldots,l}$. We then partition $L^{0,\ldots,l}$ into t disjoint, roughly equal-sized sublists $L_j^{0,\ldots,l}$, $1 \leq j \leq t$, and finally set each task T_j to perform its enumeration over the given interval but only considering the lists of chunk candidates $L_j^{0,\ldots,l}, L^{l+1}, \ldots, L^{\mathcal{N}-1}$. Note that the workload assigned to each enumerating task is a consequence of the selected method for partitioning the list $L^{0,\ldots,l}$.

Additionally, both parallelisation methods can be combined by partitioning the given interval $[B_1, B_2]$ into n_s disjoint sub-intervals and searching each such sub-interval with t_k tasks, hence amounting to $\sum_{k=1}^{n_s} t_k$ enumerating tasks.

3.2.3.6 `Threshold` Algorithm

Algorithm 9 shares some similarities with the algorithm `Threshold` introduced in [45], since `Threshold` also makes use of an array (`partialSum`) similar to the array `minArray` to speed up the pruning process. However, `Threshold` works with non-negative integer values (weights) rather than scores. `Threshold` restricts the scores to weights such that the smallest weight is the likeliest score, by making use of a function that converts scores into weights [48].

Assuming the scores have already been converted to weights, `Threshold` first sorts each list of chunk candidates $L^i = [\mathsf{c}_0^i, \mathsf{c}_1^i, \ldots, \mathsf{c}_{m_i-1}^i], 0 \leq i < \mathcal{N}$ in ascending order based on the score/weight component of its chunk candidates. It then computes the entries of

`partialSum` by first setting $\texttt{partialSum}[\mathcal{N}-1] = 0$ and then computing

$$\texttt{partialSum}[i] = \texttt{partialSum}[i+1] + \texttt{c}_0^i.score \text{ for } i = \mathcal{N}-2, \mathcal{N}-3, \ldots, 0.$$

`Threshold` then enumerates all the key candidates in a range of the form $[0, W_t)$, where W_t is a parameter. To do so, it performs a similar process to Algorithm 9 by using its precomputed table (`partialSum`) to avoid useless paths, hence improving the pruning process. This enumeration process performed by `Threshold` is described in Algorithm 10.

Algorithm 10 enumerates all key candidate in the interval $[0, W_t)$.

```
1: function THRESHOLD(i, w, K, W_t, L).
2:     for j = 0 to m_i do
3:         newW ← w + c_j^i.score;
4:         if newW + partialSum[i] > W_t then
5:             break;
6:         else
7:             if i = N − 1 then
8:                 K_i ← j;
9:                 c ← combine(c^0_{K[0]}, ..., c^{N−1}_{K[N−1]});
10:                L ← L.add(c);
11:            else
12:                K_i ← j
13:                L ← threshold(i + 1, newW, K, W_t, L)
14:            end if
15:        end if
16:    end for
17:    return L;
18: end function
```

According to its designers, this algorithm may perform a non-optimal enumeration to a depth of 2^{40}, if some adjustments are made on how the candidate keys (L) are stored. However, its primary drawback is that it must always start enumerating from the most likely key. Consequently, whilst the simplicity and relatively strong time complexity of Threshold is desirable, in a parallelised environment it can only serve as the first enumeration algorithm (or can only be used in the first search task). `Threshold` therefore was not implemented and hence is not included in the comparison made in Section 3.3.

3.2.4 A Score-Based Key Enumeration Algorithm

In this subsection, we will describe a non-optimal enumeration algorithm based on the algorithm introduced in [11]. This algorithm differs from the original algorithm in the manner in which this algorithm builds a precomputed table (`iRange`) and uses it during

execution to constructing key candidates whose total accumulated score is equal to a certain accumulated score. This algorithm shares similarities with the stack-based, depth-first key enumeration algorithm described in Section 3.2.3, because both algorithms essentially perform a depth-first search in the undirected graph G. However, this algorithm controls the pruning by the accumulated total score that a key candidate must reach to be accepted. To achieve this, the scores are restricted to positive integer values, which may be derived from a correlation value in a side-channel analysis attack.

This algorithm starts off by generating all key candidates with the largest possible accumulated total score S_1, and then proceeds to generate all key candidates whose scores are equal to the second largest possible accumulated total score S_2,[4] and so forth, until generating all key candidates with the minimum possible accumulated total score S_N. To find a key candidate whose score is equal to a certain accumulated score, this algorithm makes use of a simple backtracking strategy, which is efficient because impossible paths can be pruned early. The pruning is controlled by the accumulated score that must be reached for the solution to be accepted. To achieve a fast decision process during the backtracking, this algorithm precomputes tables for minimal and maximal accumulated total scores that can be reached by completing a path to the right, like the tables `minArray` and `maxArray` introduced in Section 3.2.3.3. Besides the tables `minArray` and `maxArray`, this algorithm also precomputes an additional table, `iRange`.

Given $0 \leq i \leq \mathcal{N}$ and $\texttt{minArray}[i] \leq s \leq \texttt{maxArray}[i]$, the entry $\texttt{iRange}[i][s]$ points to a list of integers $L^{(i,s)} = [k_0^{(i,s)}, k_1^{(i,s)} \ldots, k_n^{(i,s)}]$, where each entry represents a distinct index of the list L^i, i.e., $0 \leq k_j^{(i,s)} \neq k_u^{(i,s)} < m_i$ for $j \neq u$. The algorithm uses these indices to construct a chunk candidate with an accumulated score s from chunk i to chunk $\mathcal{N} - 1$.

In order to compute this table, we use the observation that for a given entry $k_j^{(i,s)}$ of $\texttt{iRange}[i][s]$, the list $\texttt{iRange}[i+1][cs]$, with $cs = s - \mathtt{c}^i_{k_j^{(i,s)}}.score$, must be defined and be non-empty. So we first set the entry $\texttt{iRange}[\mathcal{N}][0]$ to $[0]$ and then proceed to compute the entries for $i = \mathcal{N} - 1, \ldots, 0$ and $s = \texttt{minArray}[i], \ldots, \texttt{maxArray}[i]$. Algorithm 11 describes precisely how this table is precomputed.

[4] S_2 may not equal $S_1 - 1$.

Algorithm 11 precomputes the table `iRange`.

```
1: function PrecomputeIRange().
2:     iRange[𝒩][0] ← [0];
3:     for i = 𝒩 − 1 to 0 do
4:         for s = minArray[i] to maxArray[i] do
5:             L^(i,s) ← [];
6:             for k = 0 to m_i − 1 do
7:                 cs = s − c_k^i.score;
8:                 if iRange[i + 1][cs].size() > 0 then
9:                     L^(i,s).add(k);
10:                end if
11:            end for
12:            if L^(i,s).size() > 0 then
13:                iRange[i][s] ← L^(i,s);
14:            end if
15:        end for
16:    end for
17:    return iRange;
18: end function
```

3.2.4.1 Complete Algorithm

Algorithm 12 describes the backtracking strategy more precisely, making use of the precomputed tables for pruning impossible paths. The integer array `tscores` contains accumulated scores in a selected order, where an entry $s \in$ `tscores` must satisfy that the list `iRange`$[0][s]$ is non-empty, i.e., `iRange`$[0][s]$`.size()` > 0. This helps in constructing a key candidate with an accumulated score s from chunk 0 to chunk $\mathcal{N} - 1$. In particular, `tscores` may be set to $[S_1, S_2, \ldots, S_N]$, i.e., the array containing all possible accumulated scores that can be reached from chunk 0 to chunk $\mathcal{N} - 1$.

Furthermore, the order in which the elements in the array `tscores` are arranged is important. For this array $[S_1, S_2, \ldots, S_N]$, for example, the algorithm will first enumerate all key candidates with accumulated score S_1, then all those with accumulated score S_2 and so on. This guarantees a certain quality, i.e., good key candidates will be enumerated earlier than worse ones. However key candidates with the same accumulated score will be generated in no particular order, so a lack of precision in scores will lead to some decrease of quality.

Algorithm 12 makes use of the table `k` with $\mathcal{N}$ entries, each of which is a 2-tuple of the form (e_1, e_2) with e_1 and e_2 integers. For a given tuple `k`$[i]$, the component `k`$[i].e_1$ is an index of some list `iRange`$[i][s]$, with `minArray`$[i] \leq s \leq$ `maxArray`$[i]$, while `k`$[i].e_2$ is the corresponding value, i.e., `k`$[i].e_2 =$ `iRange`$[i][s]$`.get(`k`$[i].e_1)$. The value of `k`$[i].e_1$ allows the algorithm to control if the list `iRange`[i][s] has been traveled completely or not, while the

second component allows the algorithm to retrieve the chunk candidate of index $\mathtt{k}[i].e_2$ of L^i. This is done to avoid re-calculating $\mathtt{k}[i].e_2$ each time it is required during the execution of the algorithm.

We will now analyse Algorithm 12.

Suppose that s is set to $S \in \mathtt{tscores}$, hence $\mathtt{iRange}[0][s].\mathtt{size}() > 0$. The algorithm will then set $\mathtt{k}[0]$ to $(0, e_2^{(0)})$, with $e_2^{(0)}$ being the integer from the entry of index 0 of $\mathtt{iRange}[0][s]$, and then set cs to s (lines 3 to 5). We claim that the main **while** loop (lines 6 to 23) at each iteration will compute $\mathtt{k}[i]$ for $0 \leq i \leq \mathcal{N}-1$ such that the key candidate $\mathtt{c}$ constructed at line 12 will have an accumulated score s.

Algorithm 12 enumerates key candidates for given scores.

```
1: function KeyEnumeration(tscores,iRange).
2:     for s ∈ tscores do
3:         i ← 0;
4:         k[0] ← (0, iRange[0][s].get(0)); 2-tuple (e1, e2)
5:         cs ← s;
6:         while i ≥ 0 do
7:             while i < N − 1 do
8:                 cs ← cs − c^i_{k[i].e2}.score;
9:                 i ← i + 1;
10:                k[i] ← (0, iRange[i][cs].get(0));
11:            end while
12:            c ← combine(c^0_{k[0].e2}, ..., c^{N−1}_{k[N−1].e2});
13:            Test(c);
14:            while i ≥ 0 and k[i].e1 ≥ (iRange[i][cs].size() − 1) do
15:                i ← i − 1;
16:                if i ≥ 0 then
17:                    cs ← cs + c^i_{k[i].e2}.score;
18:                end if
19:            end while
20:            if i ≥ 0 then
21:                k[i] ← (k[i].e1 + 1, iRange[i][cs].get(k[i].e1 + 1));
22:            end if
23:        end while
24:    end for
25: end function
```

Let us set $cs_0 = s$. We know that the list $\mathtt{iRange}[0][cs_0]$ is non-empty, hence for any entry $e_2^{(0)}$ in the list $\mathtt{iRange}[0][cs_0]$, the list $\mathtt{iRange}[1][cs_1]$ is non-empty, where

$$\mathtt{minArray}[1] \leq cs_1 = cs_0 - \mathtt{c}^0_{e_2^{(0)}}.score \leq \mathtt{maxArray}[1].$$

Likewise, for any entry $e_2^{(1)}$ in the list $\mathtt{iRange}[1][cs_1]$, the list $\mathtt{iRange}[2][cs_2]$ is non-empty, where

$$\mathtt{minArray}[2] \leq cs_2 = cs_1 - \mathtt{c}^1_{e_2^{(1)}}.score \leq \mathtt{maxArray}[2].$$

Hence, for $0 \leq i < \mathcal{N}$, we have that for any given entry $e_2^{(i)}$ in the list $\texttt{iRange}[i][cs_i]$, the list $\texttt{iRange}[i+1][cs_{i+1}]$ is non-empty, where

$$\texttt{minArray}[i+1] \leq cs_{i+1} = cs_i - \texttt{c}^i_{e_2^{(i)}}.score \leq \texttt{maxArray}[i+1].$$

Note that when $i = \mathcal{N} - 1$, the list $\texttt{iRange}[i+1][0] = [0]$ is non-empty and $cs_{i+1} = 0$.

Given $\texttt{k}[0], \texttt{k}[1], \ldots, \texttt{k}[j]$ are already set for some $0 \leq j < \mathcal{N} - 1$, the first inner **while** loop (lines 7 to 11) will set $\texttt{k}[i] = (0, e_2^{(i)})$, where $e_2^{(i)}$ holds the entry of index 0 of $\texttt{iRange}[i][cs_i]$, for $0 < j < i \leq \mathcal{N} - 1$. Therefore, once the **while** loop ends, $i = \mathcal{N} - 1$ and $cs_{i+1} = cs_{\mathcal{N}} = cs_i - \texttt{c}^i_{e_2^{(i)}}.score = 0$, hence the key candidate constructed from the second components $\texttt{k}[i].e_2$ will have an accumulated score s. In particular, the first time $\texttt{k}[0]$ is set, and so the first inner **while** loop will calculate $\texttt{k}[1], \ldots, \texttt{k}[\mathcal{N} - 1]$.

Since there may be more than one key candidate with an accumulated score s, the second inner **while** loop (lines 14 to 19) will backtrack to a chunk $0 \leq i < \mathcal{N}$, from which a new key candidate with accumulated score s can be constructed. This is done by simply moving backwards (line 15) and updating cs_{i+1} to $cs_i = cs_{i+1} + \texttt{c}^i_{\texttt{k}[i].e_2}.score$ until there is an $0 \leq i < \mathcal{N}$ such that $\texttt{k}[i].e_1 < \texttt{iRange}[i][cs_i].\texttt{size}() - 1$.

- If there is such $0 \leq i < \mathcal{N}$, then the instruction at line 21 will update $\texttt{k}[i]$ to $(\texttt{k}[i].e_1 + 1, \texttt{iRange}[i][cs_i].\texttt{get}(\texttt{k}[i].e_1 + 1))$. This means that the updated value for the second component of $\texttt{k}[i]$ will be a valid index in L^i, so $\texttt{c}^i_{\texttt{k}[i].e_2}$ will be the new chunk candidate for chunk i. Then the first inner **while** loop (lines 7 to 11) will again execute and compute the indices for the remaining chunk candidates in the lists $L^{i+1}, \ldots, L^{\mathcal{N}-1}$ such that the resulting key candidate will have the accumulated score s.
- Or else, meaning that $i < 0$, then the main **while** loop (lines 6 to 23) will end and s will be set to a new value from `tscores`, since all key candidates with an accumulated score s have just been enumerated.

3.2.4.2 Parallelisation

Suppose we would like to have t tasks $T_1, T_2, T_3, \cdots, T_t$ executed in parallel to enumerate key candidates whose accumulated total scores are equal to those in the array `tscores`.

We then can split the array `tscores` into t disjoint sub-arrays $\texttt{tscores}^i$, and then set each task T_i to run Algorithm 12 through the sub-array $\texttt{tscores}^i$. As an example of a partition algorithm to distribute the workload among the tasks, we set the sub-array $\texttt{tscores}^i$ to contain elements with indices congruent to $i \mod t$ from `tscores`. Additionally, note that if we have access to the number of candidates to be enumerated for each score in the array `tscores` beforehand, we may design a partition algorithm for distributing the workload among the tasks almost evenly.

3.2.4.3 Running Times

We assume each list of chunk candidates $L^i = [\texttt{c}_0^i, \texttt{c}_1^i, \ldots, \texttt{c}_{m_i-1}^i], 0 \leq i < \mathcal{N}$, is in decreasing order based on the score component of its chunk candidates.

Regarding the running time for computing the tables `maxArray` and `minArray`, note that each entry of the table `minArray` (`maxArray`) can be computed as explained in Section 3.2.3.3. Therefore, the running time of such an algorithm is $\Theta(\mathcal{N})$.

Regarding the running time for computing `iRange`, we will analyse Algorithm 11. This algorithm is composed of three **while** blocks. For each i, $0 \leq i < \mathcal{N}$, the **while** loop from line 4 to line 15 will be executed r_i times, where $r_i = \texttt{maxArray}[i] - \texttt{minArray}[i] + 1$.

For each iteration, the innermost **for** block (lines 6 to 11) will execute simple instructions m_i times. Therefore, once the innermost **for** block has finished, its running time will be $T_3 \cdot m_i + C_3$, where T_3 and C_3 are constants. Then the **if** block (lines 12 to 14) will be attempted and its running time will be C_2, where C_2 is another constant. Therefore, the running time for an iteration of the **while** loop (lines 4 to 15) will be $T_3 \cdot m_i + C_2 + C_3$.

Therefore, the running time of Algorithm 11 is $\sum_{i=0}^{\mathcal{N}-1} r_i(T_3 \cdot m_i + C_2 + C_3)$. More specifically,

$$\sum_{i=0}^{\mathcal{N}-1} (\texttt{maxArray}[i] - \texttt{minArray}[i] + 1)(T_3 \cdot m_i + C_2 + C_3).$$

This running time depends heavily on $r_i = \texttt{maxArray}[i] - \texttt{minArray}[i] + 1$. Now the size of the range $[\texttt{minArray}[i], \texttt{maxArray}[i]]$ relies on the scaling technique used to get a positive integer from a real number. The more accurate the scaling technique is, the more

different integer scores there will be. Hence, if we use an accurate scaling technique, we will probably get larger r_i.

We will analyse the running time for Algorithm 12 to generate all key candidates whose total accumulated score is s. Let us assume there are N_s key candidates whose total accumulated score is equal to s.

First the running time for instructions at lines 3 to 5 is constant. Therefore, we will only focus on the **while** loop (lines 6 to 23).

In any iteration, the first inner **while** loop (lines 7 to 11) will execute and compute the indices for the remaining chunk candidates in the lists $L^i, \ldots, L^{\mathcal{N}-1}$, with i starting at any number in $[0, \mathcal{N}-2]$, such that the resulting key candidate will have the accumulated score s. Therefore, its running time is at most $C \cdot (\mathcal{N}-1)$, where C is a constant, i.e., it is $\mathcal{O}(\mathcal{N})$. The instruction at line 12 will combine all chunks from 0 to $\mathcal{N}-1$ and hence its running time is also $\mathcal{O}(\mathcal{N})$. The next instruction `Test`(c) will test c and its running time will depend on the scenario the algorithm is being run. Let us assume its running time is $\mathcal{O}(T(\mathcal{N}))$, where T is a function.

Regarding the second inner **while** loop (lines 14 to 19), this loop will backtrack to a chunk i with $0 \leq i < \mathcal{N}$, from which a new key candidate with accumulated score s can be constructed. This is done by simply moving backwards while computing some simple operations. Therefore, the running time for the second inner **while** loop is at most $D \cdot (\mathcal{N}-1)$, where D is a constant, i.e., it is $\mathcal{O}(\mathcal{N})$.

Therefore, the running time for generating all key candidates whose total accumulated score is s will be $\mathcal{O}(N_s \cdot (\mathcal{N} + T(\mathcal{N})))$.

3.2.4.4 Memory Consumption

Besides the precomputed tables, it is easy to see that Algorithm 12 makes use of negligible memory while enumerating key candidates. Indeed, testing key candidates is done on the fly to avoid storing them during enumeration. However, the table `iRange` may have many entries.

Let N_e be the number of entries of the table `iRange`. Line 2 of Algorithm 11 will create the entry `iRange`$[\mathcal{N}][0]$ that points to the list [0]. Hence, after the instruction at line 2 has been executed, $N_e = 1$.

Let us consider the **while** loop from line 4 to line 15. For each i, $0 \leq i < \mathcal{N}$, let S_i be the set of different values s in the range $[\texttt{minArray}[i], \texttt{maxArray}[i]]$ such that $L^{(i,s)}$ is non-empty. After the iteration for i has been executed, the table `iRange` will have $|S_i|$ new entries, each of which will point to a non-empty list, with $0 < |S_i| \leq r_i$. Therefore, $N_e = 1 + \sum_{i=0}^{\mathcal{N}-1} |S_i|$ after Algorithm 11 has completed its execution.

Note that $|S_i|$ may increase if the range $[\texttt{minArray}[i], \texttt{maxArray}[i]]$ is large. The size of this interval relies on the scaling technique used to get a positive integer from a real number. The more accurate the scaling technique is, the more different integer scores there will be. Hence, if we use an accurate scaling technique, we will probably get larger r_i, making it likely for $|S_i|$ to increase. Therefore, the table `iRange` may have many entries.

Regarding the number of bits used in memory to store the table `iRange`, let us suppose that an integer is stored in B_{int} bits and that a pointer is stored in B_p bits. Once Algorithm 11 has completed its execution, we know that `iRange`$[i][s]$ will point to the list $L^{(i,s)}$, with $0 \leq i \leq \mathcal{N}$ and $s \in S_i$. Moreover, by definition we know that the list $L^{(\mathcal{N},0)}$ will be the list [0], while any other list $L^{(i,s)}$, $0 \leq i < \mathcal{N}$ and $s \in S_i$, will have $n^{(i,s)}$ entries, with $1 \leq n^{(i,s)} \leq m_i$.

Therefore, the number of bits `iRange` occupies in memory after Algorithm 11 has completed its execution is

$$T_b = B_{int} + B_p + \sum_{i=0}^{\mathcal{N}-1} \sum_{s \in S_i} (n^{(i,s)} \cdot B_{int} + B_p). \tag{3.1}$$

Since $1 \leq n^{(i,s)} \leq m_i$, we have

$$B_{int} + B_p + \sum_{i=0}^{\mathcal{N}-1} |S_i| \cdot (B_{int} + B_p) \leq T_b \leq B_{int} + B_p + \sum_{i=0}^{\mathcal{N}-1} |S_i| \cdot (m_i \cdot B_{int} + B_p).$$

3.2.5 A Key Enumeration Algorithm using Histograms

In this subsection, we will describe a non-optimal key enumeration algorithm introduced in [61].

3.2.5.1 Setup

We now introduce a couple of tools that we will use to describe the sub-algorithms used in the algorithm of [61], using the following notations: H will denote a histogram, N_b will denote a number of bins, b will denote a bin and x a bin index.

Linear histograms. The function $H_i = \texttt{createHist}(L^i, N_b)$ creates a standard histogram from the list of chunk candidates L^i with N_b linearly-spaced bins.

Given a list of chunk candidates L^i, the function `createHist` will first calculate both the minimum score min and maximum score max among all the chunk candidates in L^i. It will then partition the interval $I = [min, max]$ into subintervals $I_0 = [min, min + \delta), I_1 = [min + \delta, min + 2\delta), \ldots, I_{N_b-1} = [min + (N_b - 1)\delta, max]$, where $\delta = \frac{max-min}{N_b}$. It then will proceed to build the list L_{H_i} of size N_b. The entry $0 \leq x < N_b$ of L_{H_i} will point to a list that contains all chunk candidates from L^i such that their scores lie in I_x. The returned standard histogram H_i is therefore stored as the list L_{H_i} whose entries will point to lists of chunk candidates. For a given bin index x, $L_{H_i}.\texttt{get}(\text{x})$ outputs the list of chunk candidates contained in the bin of index x of H_i. Therefore, $H_i[x] = L_{H_i}.\texttt{get}(x).\texttt{size}()$ is the number of chunk candidates in the bin of index x of H_i. The running time for $\texttt{createHist}(L^i, N_b)$ is $\Theta(m_i + N_b)$.

Convolution. This is the usual convolution algorithm which computes $H_{1:2} = \texttt{conv}(H_1, H_2)$ from two histograms H_1 and H_2 of sizes n_1 and n_2 respectively, where $H_{1:2}[k] = \sum_{i=0}^{k} H_1[i] \cdot H_2[k-i]$. The computation of $H_{1:2}$ is done efficiently by using Fast Fourier Transformation (FFT) for polynomial multiplication. Indeed, the array $[H_j[0], H_j[1], \ldots, H_j[n_j-1]]$ is seen as the coefficient representation of $P_j = H_j[0] + H_j[1]x + \ldots\ H_j[n_j - 1]x^{n_j-1}$ for $j = 1, 2$. In order to get $H_{1:2}$, we multiply the two polynomials of degree-bound $n = \max\{n_1, n_2\}$ in time $\Theta(n \log n)$, with both the input and output representations in coefficient form [17]. The convoluted histogram $H_{1:2}$ is therefore stored as a list of integers.

Getting the size of a histogram. The method `size`() returns the number of bins of a histogram. This method simply returns L.`size`(), where L is the underlying list used to represent the histogram.

Getting chunk candidates from a bin. Given a standard histogram H_i and an index $0 \leq x < H_i$.`size`(), the method H_i.`get`(x) outputs the list of all chunk candidates contained in the bin of index x of H_i, i.e., this method simply returns the list L_{H_i}.`get`(x).

3.2.5.2 Complete Algorithm

This key enumeration algorithm uses histograms to represent scores, and the first step of the key enumeration is a convolution of histograms modelling the distribution of the $\mathcal{N}$ lists of scores. This step is detailed in Algorithm 13.

Algorithm 13 computes standard and convoluted histograms.

function createHistograms($L^0, \ldots, L^{\mathcal{N}-1}, N_b$).
  $H_0 \leftarrow$ `createHist`(L^0, N_b);
  $H_1 \leftarrow$ `createHist`(L^1, N_b);
  $H_{0:1} \leftarrow$ `conv`(H_0, H_1);
  for $i = 2$ to $\mathcal{N} - 1$ **do**
    $H_i \leftarrow$ `createHist`(L^i, N_b);
    $H_{0:i} \leftarrow$ `conv`($H_i, H_{0:i-1}$);
  end for
  return $H = [H_0, H_1, \ldots, H_{\mathcal{N}-1}, H_{0:1}, \ldots, H_{0:\mathcal{N}-1}]$;
end function

Algorithm 14 computes the indices' bounds.

function computeBounds(R_1,R_2).
  $start \leftarrow H_{0:\mathcal{N}-1}$.`size`();
  $cnt_{start} \leftarrow 0$;
  while $cnt_{start} < R_1$ **do**
    $start \leftarrow start - 1$;
    $cnt_{start} \leftarrow cnt_{start} + H_{0:\mathcal{N}-1}[start]$;
  end while
  $x_{start} \leftarrow start$;
  while $cnt_{start} < R_2$ **do**
    $start \leftarrow start - 1$;
    $cnt_{start} \leftarrow cnt_{start} + H_{0:\mathcal{N}-1}[start]$;
  end while
  $x_{stop} \leftarrow start$;
  return x_{start}, x_{stop};
end function

Based on this first step, this key enumeration algorithm allows enumerating key candidates that are ranked between two bounds R_1 and R_2. In order to enumerate all keys ranked

between the bounds R_1 and R_2, the corresponding indices of bins of $H_{0:\mathcal{N}-1}$ have to be computed, as described in Algorithm 14. It simply sums the number of key candidates contained in the bins starting from the bin containing the highest scoring key candidates, until we exceed R_1 and R_2, and returns the corresponding indices x_{start} and x_{stop}.

Algorithm 15 performs bin decomposition.

```
1: function decomposeBin(H,csh,x_bin,kf)
2:     if csh = 1 then
3:         x ← H_0.size() − 1;
4:         while (x ≥ 0) and ((x + H_1.size()) ≥ x_bin); do
5:             if H_0[x] > 0 and H_1[x_bin − x] > 0 then
6:                 kf(0) ← H_0.get(x);
7:                 kf(1) ← H_1.get(x_bin − x);
8:                 processKF(kf);
9:             end if
10:            x ← x − 1
11:        end while
12:    else
13:        x ← H_csh.size() − 1;
14:        while (x ≥ 0) and ((x + H_0:csh−1.size()) ≥ x_bin) do
15:            if H_csh[x] > 0 and H_0:csh−1[x_bin − x] > 0 then
16:                kf(csh) ← H_csh.get(x);
17:                decomposeBin(H, csh − 1, x_bin − x, kf);
18:            end if
19:            x ← x − 1;
20:        end while
21:    end if
22: end function
```

Algorithm 16 processes table `kf`.

```
1: function processKF(kf)
2:     i ← 0;
3:     I[i] ← 0;
4:     while i ≥ 0 do
5:         while i < N − 1 do
6:             i ← i + 1;
7:             I[i] ← 0;
8:         end while
9:         c ← combine(kf[0].get(I[0]), ..., kf[N − 1].get(I[N − 1]));
10:        Test(c);
11:        while i ≥ 0 and I[i] ≥ (kf[i].size() − 1) do
12:            i ← i − 1;
13:        end while
14:        if i ≥ 0 then
15:            I[i] ← I[i] + 1;
16:        end if
17:    end while
18: end function
```

Given the list of histograms of scores H and the indices of bins of $H_{0:\mathcal{N}-1}$ between which we want to enumerate, the enumeration simply consists of performing a backtracking over all the bins between x_{start} and x_{stop}. More precisely, during this phase we recover the bins of the initial histograms (i.e. before convolution) that were used to build a bin of

the convoluted histogram $H_{0:\mathcal{N}-1}$. For a given bin b with index x of $H_{0:\mathcal{N}-1}$, we have to run through all the non-empty bins $b_0, \ldots, b_{\mathcal{N}-1}$ of indices $x_0, \ldots x_{\mathcal{N}-1}$ of $H_0, \ldots, H_{\mathcal{N}-1}$ such that $x_0 + \ldots + x_{\mathcal{N}-1} = x$. Each b_i will then contain at least one and at most m_i chunk candidates of the list L^i that we must enumerate. This leads to storing a table `kf` of $\mathcal{N}$ entries, each of which points to a list of chunk candidates. The list pointed to by the entry `kf`$[i]$ holds at least one and at most m_i chunk candidates contained in the bin b_i of the histogram H_i. Any combination of these $\mathcal{N}$ lists, i.e., picking an entry from each list, results in a key candidate.

Algorithm 15 describes more precisely this bin decomposition process. This algorithm simply follows a recursive decomposition. That is, in order to enumerate all the key candidates within a bin b of index x of $H_{0:\mathcal{N}-1}$, it first finds two non-empty bins of indices $x_{\mathcal{N}-1}$ and $x - x_{\mathcal{N}-1}$ of $H_{\mathcal{N}-1}$ and $H_{0:\mathcal{N}-2}$ respectively. All the chunk candidates in the bin of index $x_{\mathcal{N}-1}$ of $H_{\mathcal{N}-1}$ will be added to the key factorisation, i.e., the entry `kf`$[\mathcal{N}-1]$ will point to the list of chunk candidates returned by $H_{\mathcal{N}-1}$.`get`$(x_{\mathcal{N}-1})$. It then continues the recursion with the bin of index $x - x_{\mathcal{N}-1}$ of $H_{0:\mathcal{N}-2}$ by finding two non-empty bins of indices $x_{\mathcal{N}-2}$ and $x - x_{\mathcal{N}-1} - x_{\mathcal{N}-2}$ of $H_{\mathcal{N}-2}$ and $H_{0:\mathcal{N}-3}$ respectively and adding all the chunk candidates in the bin of index $x_{\mathcal{N}-2}$ of $H_{\mathcal{N}-2}$ to the key factorisation, i.e., `kf`$[\mathcal{N}-2]$ will now point to the list of chunk candidates returned by $H_{\mathcal{N}-2}$.`get`$(x_{\mathcal{N}-2})$, and so forth. Eventually each time a factorisation is completed, Algorithm 15 calls the function `processKF` which takes as input the table `kf`. The function `processKF`, as defined in Algorithm 16, will compute the candidate keys from `kf`. This algorithm basically generates all the possible combinations from the $\mathcal{N}$ lists `kf`$[i]$. Note that this algorithm may be seen as a particular case of Algorithm 12.

Finally, the main loop of this key enumeration algorithm simply calls Algorithm 15 for all the bins of $H_{0:\mathcal{N}-1}$ which are between the enumeration bounds x_{start}, x_{stop}.

3.2.5.3 Parallelisation

Suppose we would like to have t tasks $T_1, T_2, T_3, \cdots, T_t$ executing in parallel to enumerate key candidates that are ranked between two bounds R_1 and R_2 in parallel. We can then calculate the indices x_{start}, x_{stop}, and then create the array $\mathtt{X} = [x_{start}, x_{start} - 1, \ldots, x_{stop}]$. We then partition the array `X` into t disjoint sub-arrays $\mathtt{X}_i$, and finally set each task T_i to call the function `decomposeBin` for all the bins of $H_{0:\mathcal{N}-1}$ with indices in $\mathtt{X}_i$.

As has been noted previously, the algorithm employed to partition the array `X` directly allows efficient parallel key enumeration, where the amount of computation performed by each task may be well-balanced. As an example of a partition algorithm that could almost evenly distribute the workload among the tasks is:

1. Set i to 0.
2. If `X` is non-empty, pick an index x in `X` such that $H_{0:\mathcal{N}-1}[x]$ is the maximum number. Or else return $\mathtt{X}_0, \mathtt{X}_1, \ldots, \mathtt{X}_t$.
3. Remove x from the array `X` and add it to the array $\mathtt{X}_i$.
4. Update i to $(i+1) \mod t$ and go back to Step 2.

3.2.5.4 Memory Consumption

Besides the precomputed histograms, which are stored as arrays in memory, it is easy to see that this algorithm makes use of negligible memory (only the table `kf`) while enumerating key candidates. Additionally, it is important to note that each time the function `processKF` is called, it will need to generate all key candidates obtained by picking chunk candidates from the $\mathcal{N}$ lists pointed to by the entries of `kf` and process all of them immediately, since the table `kf` may have changed. This implies that if the processing of key candidates is left to be done after the complete enumeration has finished, each version of the table `kf` would need to be stored, which again might be problematic in terms of memory consumption.

Regarding how many bits in memory the precomputed histograms consumes, we will analyse Algorithm 13.

First note for a given list of chunk candidates L^i and N_b, the function `createHist`(L^i, N_b) will return the standard histogram H_i. This standard histogram will be stored as the list L_{H_i} of size N_b. An entry x of L_{H_i} will point to a list of chunk candidates. The total number of chunk candidates hold by all the lists pointed to by the entries of L_{H_i} is m_i. Therefore, the number of bits to store the list L_{H_i} is $B_p \cdot N_b + B_c \cdot m_i$, where B_p is the number of bits to store a pointer and B_c is the number of bits to store a chunk candidate

$(score, [e])$. The total number of bits to store all lists L_{H_i}, $0 \leq i < \mathcal{N}$, is

$$\sum_{i=0}^{\mathcal{N}-1} (B_p \cdot N_b + B_c \cdot m_i) = \mathcal{N} \cdot B_p \cdot N_b + B_c \cdot \sum_{i=0}^{\mathcal{N}-1} m_i. \tag{3.2}$$

Concerning the convoluted histograms, let us first look at $H_{0:1} = \texttt{conv}(H_0, H_1)$. We know that $H_{0:1}$ is stored as a list of integers and that these entries can be seen as the coefficients of the resulting polynomial from multiplying the polynomial $P_0 = H_0[0] + H_0[1]x + \ldots\ H_0[N_b-1]x^{N_b-1}$ by $P_1 = H_1[0] + H_1[1]x + \ldots\ H_1[N_b-1]x^{N_b-1}$. Therefore, the list of integers used to store $H_{0:1}$ has $2N_b - 1$ entries. Following a similar reasoning to the previous one, we can conclude that the list of integers used to store $H_{0:2} = \texttt{conv}(H_2, H_{0:1})$ has $3N_b - 2$ entries. Therefore, for a given i, $1 \leq i \leq \mathcal{N} - 1$, the list of integers used to store $H_{0:i} = \texttt{conv}(H_i, H_{0:i-1})$ has $(i+1)N_b - i$ entries. The total number of entries of all the convoluted histograms $H_{0:1}, H_{0:2}, \ldots, H_{0:\mathcal{N}-1}$ is

$$\sum_{i=1}^{\mathcal{N}-1} ((i+1)N_b - i) = (N_b - 1)\frac{(\mathcal{N}-1)(\mathcal{N})}{2} + N_b(\mathcal{N}-1).$$

As expected, the total number of entries strongly depends on the values N_b and $\mathcal{N}$. If an integer is stored in B_{int} bits, then the number of bits for storing all the convoluted histograms is

$$B_{int} \cdot (N_b - 1)\frac{(\mathcal{N}-1)(\mathcal{N})}{2} + B_{int} \cdot N_b(\mathcal{N}-1). \tag{3.3}$$

3.2.5.5 Equivalence with the path counting approach

The stack-based key enumeration algorithm and the score-based key enumeration algorithm can be also used for rank computation (instead of enumerating each path, the rank version counts each path). Similarly, the histogram algorithm can also be used for rank computation by simply summing the size of the corresponding bins in $H_{0:\mathcal{N}-1}$. These two approaches were believed to be distinct from each other. However, Martin *et al.* in [49] show that both approaches are mathematically equivalent, i.e., they both compute the exact same rank when choosing their discretisation parameter correspondingly. Particularly, the authors show that the binning process in the histogram algorithm is equivalent to the

"map to weight" float-to-integer conversion used prior to their path counting algorithm (Forest) by choosing the algorithms' discretisation parameter carefully. Additionally, in this paper, a performance comparison between their enumeration versions was carried out. The practical experiments indicated that Histogram performs best for low discretisation, and Forest wins for higher parameters.

3.2.6 A Quantum Key Search Algorithm

In this subsection, we will describe a quantum key enumeration algorithm introduced in [50] for the sake of completeness. This algorithm is constructed from a non-optimal key enumeration algorithm, which uses the key rank algorithm given by Martin *et al.* in [48] to return a single candidate key (the r^{th}) with a weight in a particular range. We will first describe the key rank algorithm. This algorithm restricts the scores to positive integer values (weights) such that the smallest weight is the likeliest score, by making use of a function that converts scores into weights [48].

Algorithm 17 creates the matrix $\mathtt{b}$.

```
1: function INITIALISE(W1,W2)
2:     i ← N − 1;
3:     b ← [[0]^W2]^N
4:     for w = 0 to W2 − 1 do
5:         for j = 0 to m_i − 1 do
6:             if W1 − w ≤ c_j^i.score < W2 − w then
7:                 b_{i,w} ← b_{i,w} + 1;
8:             end if
9:         end for
10:    end for
11:    for i = N − 2 to 0 do
12:        for w = 0 to W2 − 1 do
13:            for j = 0 to m_i − 1 do
14:                if w + c_j^i.score < W2 then
15:                    b_{i,w} ← b_{i,w} + b_{i+1,w+c_j^i.score};
16:                end if
17:            end for
18:        end for
19:    end for
20:    return b
21: end function
```

Assuming the scores have already been converted to weights, the rank algorithm first constructs a matrix $\mathtt{b}$ with size of $\mathcal{N} \times W_2$ for a given range $[W_1, W_2)$ as follows. For $i = \mathcal{N} - 1$ and $0 \leq w < W_2$, the entry $\mathtt{b}_{i,w}$ contains the number of chunk candidates such that their total score plus w lies in the given range. Therefore, $\mathtt{b}_{i,w}$ is given by the number of chunk candidates $\mathtt{c}^i_j, 0 \leq j < m_i$, such that $W_1 - w \leq \mathtt{c}^i_j.score < W_2 - w$.

On the other hand, for $i = \mathcal{N}-2, \mathcal{N}-3, \ldots, 0$, and $0 \leq w < W_2$, the entry $\mathtt{b}_{i,w}$ contains the number of chunk candidates that can be constructed from the chunk i to the chunk $\mathcal{N}-1$ such that their total score plus w lies in the given range. Therefore, $\mathtt{b}_{i,w}$ may be calculated as follows. For $0 \leq j < m_i$, $\mathtt{b}_{i,w} = \mathtt{b}_{i,w} + \mathtt{b}_{i+1,w+\mathtt{c}_j^i.score}$ if $w + \mathtt{c}_j^i.score < W_2$.

Algorithm 17 describes precisely the manner in which the matrix `b` is computed. Once the matrix `b` is computed, the rank algorithm will calculate the number of key candidates in the given range by simply returning $\mathtt{b}_{0,0}$. Note that $\mathtt{b}_{0,0}$, by construction, contains the number of chunk candidates, with initial weight 0, that can be constructed from the chunk 0 to the chunk $\mathcal{N}-1$ such that their total score lies in the given range. Algorithm 18 describes the rank algorithm.

Algorithm 18 returns the number of key candidates in a given range.

```
1: function RANK(W1,W2)
2:     b ← initialises(W1, W2);
3:     return b0,0;
4: end function
```

With the help of the rank algorithm, an algorithm for requesting particular candidate keys is introduced. This "getkey" algorithm returns the r^{th} key candidate with weight between W_1 and W_2. Algorithm 19 describes precisely the "getkey" algorithm. Note that the correctness of the function `getKey` follows from the correctness of `b`. Also, the algorithm is deterministic, therefore given the same r, it will return the same key candidate `k`[5].

Equipped with the "getkey" algorithm, the authors of [50] introduced a non-optimal key enumeration algorithm to enumerate and test all key candidates in the given range. This algorithm works by calling the function `getKey` to obtain a key candidate in the given range, until there are no more key candidates in the given range. Also, for each obtained key candidate `k`, it is tested by using a testing function `T` returning either 1 or 0. Algorithm 20 precisely describes how this non-optimal key enumeration algorithm works.

Combining together the function `keySearch` with the techniques for searching over partitions independently, the authors of [50] introduced a key search algorithm, described in Algorithm 21. The function `KS` works by partitioning the search space into sections whose size follows a geometrically increasing sequence using a size parameter $a = \mathcal{O}(1)$. This parameter is chosen such that the number of loop iterations is balanced with the number of keys verified per block.

[5]The r^{th} key candidate does not have to be the r^{th} most likely key in the given range.

Algorithm 19 returns the r^{th} key candidate with weight between W_1 and W_2..

```
1: function GETKEY(b, W_1, W_2, r)
2:     if r > b_{0,0} then
3:         return ⊥;
4:     end if
5:     k ← [0]^{𝒩};
6:     w ← 0;
7:     for i = 0 to 𝒩 − 2 do
8:         for j = 0 to m_i − 1 do
9:             if r < b_{i+1,w+c^i_j.score} then
10:                k_i ← j;
11:                w ← w + c^i_j.score;
12:                break j;
13:            end if
14:            r ← r − b_{i+1,w+c^i_j.score};
15:        end for
16:    end for
17:    i ← 𝒩 − 1;
18:    for j = 0 to m_i − 1 do
19:        v ← (W_1 − w ≤ c^i_j.score < W_2 − w)?1 : 0;
20:        if r ≤ v then
21:            k_i ← j;
22:            break j;
23:        end if
24:        r ← r − v;
25:    end for
26:    return k;
27: end function
```

Algorithm 20 enumerates and tests key candidates with weight between W_1 and W_2.

```
1: function KEYSEARCH(W_1, W_2, T)
2:     b ← initialises(W_1, W_2);
3:     r ← 1;
4:     while True do
5:         k ← getKey(b, W_1, W_2, r);
6:         if k =⊥ then
7:             break;
8:         end if
9:         if T(k) = 1 then
10:            break;
11:        end if
12:        r ← r + 1;
13:    end while
14:    return k;
15: end function
```

Algorithm 21 searches key candidates in a range with a size of e approximately.

```
1: function KS(e, T)
2:     W_1 ← W_min;
3:     W_2 ← W_min + 1;
4:     step ← 0;
5:     Choose W_e such that rank(0,W_e) is approx e;
6:     while W_1 ≤ W_e do
7:         k ← keySearch(W_1, W_2, T);
8:         if k ≠⊥ then
9:             return k;
10:        end if
11:        step ← step + 1;
12:        W_1 ← W_2
13:        Choose W_2 such that rank(W_1,W_2) is approx a^step;
14:    end while
15:    return k;
16: end function
```

Having introduced the function KS, the authors of [50] transformed it into a quantum key search algorithm that heavily relies on Grover's algorithm [26]. This is a quantum algorithm to solve the following problem: Given a a black box function which returns 1 on a single input x, and 0 on all other inputs, find x. Note that if there are N possible inputs to the black box function, the classical algorithm uses $\mathcal{O}(N)$ queries to the black box function since the correct input might be the very last input tested. However, in a quantum setting, a version of Grover's algorithm solves the problem using $\mathcal{O}(N^{1/2})$ queries, with certainty [26, 27]. Algorithm 22 describes the quantum search algorithm, which achieves a quadratic speed-up over the classical key search (Algorithm 21) [50]. However, it would require significant quantum memory and a deep quantum circuit, making its practical application in the near future rather unlikely.

Algorithm 22 performs a quantum search of key candidates in a range with a size of e approximately

```
1: function KQS(e, T)
2:     W_1 ← W_min;
3:     W_2 ← W_min + 1;
4:     step ← 0;
5:     Choose W_e such that rank(0,W_e) is approx e;
6:     while W_1 ≤ W_e do
7:         b ← intitialise(W_1, W_2);
8:         f(·) ← T(getKey(b, W_1, W_2, ·));
9:         Call Grover using f for one or zero marked elements in range [W_1, W_2)
10:        if marked element t found then
11:            return getKey(b, W_1, W_2, t);
12:        end if
13:        step ← step + 1;
14:        W_1 ← W_2
15:        Choose W_2 such that rank(W_1,W_2) is approx a^step;
16:    end while
17:    return k;
18: end function
```

3.3 Comparison of Key Enumeration Algorithms

In this section, we will make a comparison of the previously described algorithms. We will show some results regarding their overall performance by computing some measures of interest.

3.3.1 Implementation

All the algorithms discussed in this chapter were implemented in Java. This is because the Java platform provides the Java Collections Framework to handle data structures, which reduces programming effort, increases speed of software development and quality, and is reasonably performant. Furthermore, the Java platform also easily supports concurrent programming, providing high-level concurrency APIs.

3.3.2 Scenario

In order to make a comparison, we will consider a common scenario in which we will run the key enumeration algorithms to measure their performance. Particularly, we generate a random secret key encoded as a bit string of 128 bits, which is represented as a concatenation of 16 chunks, each on 8 bits.

We use a bit-flipping model, as described in Section 2.3. We particularly set α and β to particular values, namely 0.01 and 0.01 respectively. We then create an original key `k` (AES key) by picking a random value for each chunk i, where $0 \leq i < 16$. Once this key `k` has been generated, its bits will be flipped according to the values α, β to obtain a noisy version of it, `r`. We then use the procedure described in Secion 2.5 to assign a score to each of the 256 possible candidate values for each chunk i. Therefore, once this algorithm has ended its execution, there will be 16 lists, each having 256 chunk candidates.

These 16 lists are then given to an auxiliary algorithm that does the following. For $0 \leq i < 16$, this algorithm outputs 2^e, with $1 \leq e \leq 8$, chunk candidates for the chunk i, ensuring that the original chunk candidate for this chunk is one of the 2^e chunk candidates. This is, the secret key `k` is one out of all the $2^{16 \cdot e}$ key candidates. Therefore, we finally have access to 16 lists, each having 2^e chunk candidates, on which we run each of the key

enumeration algorithms. Additionally, on execution, the candidate keys generated by a particular key enumeration algorithm are not "tested", but rather "verified" by comparing them to the known key. Note that this is done only for the sake of testing these algorithms, however, in practice, it may be not possible to have such an auxiliary algorithm and the candidate keys have to be tested, rather than verified.

3.3.3 Results per algorithms

In order to measure the key enumeration algorithms' overall performance, we simply generate multiple random instances of the scenario. Once a random instance has been generated, each key enumeration algorithm is run for a fixed number of key candidates. For each run of any algorithm, some statistics are collected, particularly the elapsed time to enumerate a fixed number of key candidates. This was done on a machine with an Intel Xeon CPU E5-2667 v2 running at 3.30GHz with 8 cores. The set of simulations are run by setting e to 3. Therefore, each list has a size of 8 chunk candidates:

By running the optimal key enumeration algorithm (OKEA) from Section 3.2.1, we find the following issues: it is only able to enumerate at most 2^{30} key candidates; and its overall performance decreases as the number of key candidates to enumerate increases. In particular, the number of key candidates considered per millisecond per core ranges from 2336 in a 2^{20} enumeration, through 1224 in a 2^{25} enumeration, to 582 in a 2^{30} key enumeration. The main reason for this is that its memory usage grows rapidly as the number of key candidates to generate increases. Indeed, using terminology from Section 3.2.1.4, we have $W = 128 = 2^7$, $w = 8 = 2^3$, so $a = 3, b = 4$, so this instance of OKEA creates a tree composed of the root node $\mathtt{R}$, the internal nodes $\mathtt{N}_{\lambda}^{i_d}$, for $0 < \lambda \leq 3$, $0 \leq i_d < 2^{\lambda}$, and the leaf nodes $\mathtt{L}^i$, for $0 \leq i < 16$.

A chunk candidate is a 2-tuple of the form $(score, value)$, where *score* is a `float` and *value* is an `integer` array. Both a `float` variable and an `integer` variable are stored in 32 bits. Now, at level 4, *value* has only an entry, therefore $B_4 = 32 + 32 = 64$. At level 3, *value* has 2 entries, therefore $B_3 = 32 + 2(32) = 96$. At level 2, *value* has 4 entries, therefore $B_2 = 5(32) = 160$. And at level 1, *value* has 8 entries, therefore $B_1 = 9(32) = 288$. After N key candidates has been generated, the number of bits M_N used to store chunk

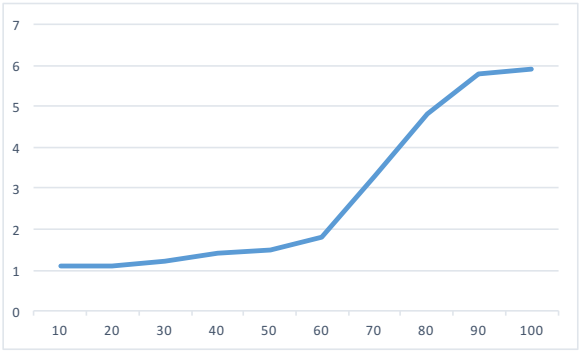


Figure 3.5: Running Times of Algorithm 13 of KEA with histograms from Section 3.2.5,. The y-axis represents the running time (milliseconds), while the x-axis represents N_b.

candidates by the algorithm will be

$$\begin{aligned} M_N =& \sum_{\lambda=1}^{3} 2^{\lambda} B_{\lambda} + B_4(\sum_{d=0}^{15} 8) + \sum_{d=1}^{N}\sum_{\lambda=1}^{3} p_{\lambda}^{(d)} B_{\lambda} \\ =& 2 \cdot B_1 + 2^2 \cdot B_2 + 2^3 \cdot B_3 + (15 \cdot 8) \cdot B_4 + \sum_{d=1}^{N} (p_1^{(d)} \cdot B_1 + p_2^{(d)} \cdot B_2 + p_3^{(d)} \cdot B_3) \\ =& 2 \cdot 288 + 2^2 \cdot 160 + 2^3 \cdot 96 + 120 \cdot 64 + \sum_{d=1}^{N} (p_1^{(d)} \cdot 288 + p_2^{(d)} \cdot 160 + p_3^{(d)} \cdot 96) \\ =& 9664 + \sum_{d=1}^{N} (p_1^{(d)} \cdot 288 + p_2^{(d)} \cdot 160 + p_3^{(d)} \cdot 96). \end{aligned}$$

Since $1 \leq p_{\lambda}^{(d)} \leq 2^{\lambda}$, for $1 \leq \lambda \leq 3, 1 \leq d \leq N$, then

$$9664 + 544 \cdot N \leq 9664 + \sum_{d=1}^{N} (p_1^{(d)} \cdot 288 + p_2^{(d)} \cdot 160 + p_3^{(d)} \cdot 96) \leq 9664 + 1984 \cdot N.$$

We also need to include the number of bits used to store extended candidates internally in each priority queue $\mathtt{N}_{\lambda}^{i_d}\mathtt{.Q}$, for $0 < \lambda \leq 3$, $0 \leq i_d < 2^{\lambda}$, and the priority queue $\mathtt{R.Q}$. Therefore, we conclude that, despite all the efforts made for implementing this algorithm in an ingenious way, the algorithm's scalability is mostly affected by its inherent design rather than by a particular implementation.

On the other hand, the bounded-space key enumeration algorithm (BSKEA) with $\omega = 4$, described in Section 3.2.2, is able to enumerate 2^{30}, 2^{33}, 2^{36} key candidates. However, it has a dramatic decrease in its overall performance as the number of key candidates to

Parameter N_b	Number of bins of $H_{0:\mathcal{N}-1}$	Total number of key candidates for $R_1 = 1$ and $R_2 = 2^{30}$
10	145	1412497166
20	224	1310161019
30	305	1260927932
40	384	1228979005
50	464	1207956426
60	545	1191780722
70	625	1178891769
80	705	1169493889
90	784	1162092971
100	864	1156185368

Table 3.1: Variation of the number of key candidates in KEA with histograms (3.2.5).

enumerate increases, similar to OKEA's behaviour. In particular, it is able to enumerate about 4800 key candidates per millisecond per core on average in a 2^{30} enumeration, but this value drops to about 1820 key candidates on average in a 2^{36} enumeration. The possible reasons for this behaviour are its intrinsic design, its memory consumption and its implementation. The variant of the bounded-space key enumeration algorithm, introduced in Section 3.2.2.4, has the same problem as OKEA, i.e., its overall performance (hence its scalability) is degraded by its excessive memory consumption and it is only able to enumerate up to 2^{30} key candidates.

Regarding the key enumeration algorithm using histograms from Section 3.2.5, we first analyse its pre-computation algorithms, i.e., Algorithm 13 and Algorithm 14. These two algorithms were run for $N_b = 10, 20, \ldots, 100$, $R_1 = 1$ and $R_2 = 2^{30}$ for 100 times. We notice that the running time increases as N_B increases, especially for Algorithm 13 as Figure 3.5 shows. On the other hand, Algorithm 14 shows some negligible variations in its running time. Besides, as expected, we note that the parameter N_b makes the number of bins of $H_{0:\mathcal{N}-1}$ increases, therefore setting this parameter to a proper value helps in guaranteeing the number of key candidates to enumerate while running through the enumeration bounds x_{start}, x_{stop} will be closer to $R_2 - R_1 + 1 = 2^{30} = 1073741824$. Table 3.1 shows the number of bins of $H_{0:\mathcal{N}-1}$ and the total number of key candidates to be enumerated between bounds x_{start}, x_{stop} on average.

Parameter N_b	Standard histograms	Convoluted histograms	Total number of bits
10	13312	39360	52672
30	23552	125760	149312
50	33792	212160	245952
70	44032	298560	342592
100	59392	428160	487552

Table 3.2: Number of bits for storing histograms in KEA with histograms (3.2.5).

Concerning the memory consumed by the arrays used to store histograms, we know that the total number of bits to store all lists L_{H_i}, $0 \leq i < 16$ is given by Equation (3.2) from Section 3.2.5.4. Therefore, we set B_p, which is the number of bits to store a pointer, to 32 bits and set B_c, the number of bits to store a chunk candidate $(score, value)$,[6] to 64. Therefore,

$$\mathcal{N} \cdot B_p \cdot N_b + B_c \cdot \sum_{i=0}^{\mathcal{N}-1} m_i = 16 \cdot B_p \cdot N_b + (128) \cdot B_c = 512 \cdot N_b + 8192. \tag{3.4}$$

Now the number of bits for storing all the convoluted histograms is given by Equation (3.3) from Section 3.2.5.4. We set $B_{int} = 32$, therefore

$$32 \cdot (N_b - 1)\frac{(15)(16)}{2} + (32 \cdot 15) \cdot N_b = 3840 \cdot (N_b - 1) + 480 \cdot N_b. \tag{3.5}$$

Table 3.2 shows the number of bits for storing both standard histograms and convoluted histograms for values $N_b = 10, 30, 50, 70, 100$.

We will describe results related to the enumeration algorithm of KEA with histograms, i.e., Algorithm 15. To run this algorithm, we first set the parameter R_1 to 1, R_2 to 2^z, where $z = 30, 33, 36$, and N_b to 60. Once the pre-computation algorithms have ended their execution, we then run Algorithm 15 for each index bin in the range calculated by Algorithm 14. As a result, we find that this algorithm is able to enumerate 2^{30}, 2^{33}, 2^{36} key candidates and that its enumeration rate is between 3500 and 3800 key candidates per millisecond per core. Additionally, as seen, its memory consumption is low.

Concerning the stack-based key enumeration algorithm from Section 3.2.3, we first calcu-

[6] The array *value* only has an entry.

late appropriate values for B_1 and B_2 by making use of the convoluted histogram $H_{0:\mathcal{N}-1}$ output by Algorithm 13. We then run Algorithm 9 with parameters B_1 and B_2, but limiting the enumeration over this interval to not exceed the number of key candidates to enumerate;[7] this number is obtained from the previous enumeration. As a result, we find that this algorithm is able to enumerate 2^{30}, 2^{33}, 2^{36} key candidates and that its enumeration rate is between 3300 and 3500 key candidates per millisecond per core.

Concerning its memory consumption, the stack-based key enumeration algorithm only uses two precomputed arrays `minArray` and `maxArray`, both of which have $\mathcal{N}+1 = 17$ `double` entries. Additionally, as pointed out in Section 3.2.3.4, at any stage of the algorithm, there are at most 16 4-tuples stored in the stack `S`. Note that a 4-tuple consists of a `double` entry, two `int` entries and an entry holding an `int` array `indices`. This array `indices` may have at most 16 entries, each holding an integer value. Therefore, its memory consumption is low.

Lastly, concerning the score-based key enumeration algorithm from Section 3.2.4, we first run its precomputation algorithms, i.e., the algorithms for computing the tables `minArray`, `maxArray` and `iRange`. As was pointed out in Sections 3.2.4.3 and 3.2.4.4, the size of table `iRange`, hence the running time for calculating it, depends heavily on the scaling technique used to get a positive integer from a real number (score). We particularly use $score \cdot 10^s$ with $s = 4$ to get an integer score from a real-valued score. We find that the table `iRange` has around 15066 entries on average. Each of these entries point to a list of integers whose number of entries is about 4 on average. Therefore, from Equation (3.1) introduced in Section 3.2.4.4, we have that the number of bits to store this table is $64 + (32 \cdot 5)(15066) = 2410624$ on average.[8] Furthermore, we run Algorithm 12, but limiting it to not exceed the number of key candidates to enumerate. As a result, we find that this algorithm is able to enumerate between 2600 and 3000 key candidates per millisecond per core.

3.3.4 Discussion

From our previous results, it can be seen that all key enumeration algorithms except for the optimal key enumeration algorithm (OKEA) and the variant of BSKEA have a much

[7] Interval $[B_1, B_2]$ may have more key candidates than the number of key candidates to enumerate.
[8] A pointer is stored in 32 bits.

Algorithm Name	Parallelizable	Memory Consumption	Scalabilty
Optimal KEA (3.2.1)	No	High	Low
Bounded-space KEA (3.2.2)	Yes, but loses near-optimality	Moderate	Moderate
Stack-based KEA (3.2.3)	Yes	Low	High
Score-based KEA (3.2.4)	Yes	Low	High
KEA with histograms (3.2.5)	Yes	Low	High

Table 3.3: Qualitative and functional attributes of key enumeration algorithms.

better overall performance and are able to enumerate a higher number of key candidates. In particular, we find that all of them are able to enumerate 2^{30}, 2^{33}, 2^{36} key candidates, while OKEA and the variant of BSKEA only are able to enumerate up to 2^{30}. Their poor performance is caused by their excessive consumption of memory. In particular, OKEA is the most memory-consuming algorithm, hence degrading its overall performance and scalability. In general, scalability is low in optimal key enumeration algorithms [68, 69] considering that not too many candidates can be enumerated, as a result of the exponential growth in their memory consumption. However, by relaxing the restriction on the order in which the key candidates will be enumerated, we are able to design non-optimal key enumeration algorithms, having a better overall performance and scalability. In particular, relaxing this restriction on the order allows for the construction of parallelizable and memory-efficient key enumeration algorithms, as was evinced in this chapter and the results previously described. Moreover, all the algorithms save OKEA [11, 48, 47, 61, 18] as described in this chapter are non-optimal ones and their respective descriptions and empirical results show that they are expected to have a better overall performance and consume much less computational resources. Table 3.3 briefly summarises some quality and functional attributes of the described algorithms.

3.4 Chapter Conclusions

In this chapter, we investigated the key enumeration problem, since there is a connection between the key enumeration problem and the key recovery problem. The key enumeration problem arises in the side-channel attack literature, where, for example, the attacker might procure scoring information for each byte of an AES key from a power analysis attack and then want to efficiently enumerate and test a large number of complete 16-byte candidates

until the correct key is found.

In summary, we first stated the key enumeration problem in a general way and then studied and analysed several algorithms to solve this problem, such as the optimal key enumeration algorithm (OKEA), the bounded-space near-optimal key enumeration algorithm, the simple stack-based, depth-first key enumeration algorithm, the score-based key enumeration algorithm, and the key enumeration algorithm using histograms. For each studied algorithm, we described its inner functioning, showing its functional and qualitative features, such as memory consumption, amenability to parallelisation and scalability. Furthermore, we proposed variants of some of them and implemented all of them on Java. We then experimented with them and made an experimental comparison of all of them, drawing special attention to their strengths and weaknesses.

Chapter 4

Cold Boot Attack on NTRU

Contents

4.1 Introduction 74
4.2 NTRU Encryption Scheme 76
4.3 Private Key Formats for NTRU Implementations 78
4.4 Mounting Cold Boot Key Recovery Attacks 82
4.5 Experimental Evaluation 87
4.6 Chapter Conclusions 94

In this chapter, we analyze the feasibility of cold boot attacks against the NTRU public key encryption scheme by reviewing two specific implementations: the first is `ntru-crypto`*, which is a pair of C and Java libraries developed by OnBoard Security. The second is* `tbuktu`*, which is a pair of libraries developed by "Tim Buktu" and is available in 'C' and Java languages. We propose a general key recovery strategy that is adapted to each one of the in-memory private key formats.*

4.1 Introduction

We examine the feasibility of cold boot attacks against the NTRU public key encryption scheme [32, 31]. We believe this to be the first time that this has been attempted. Our work can be seen as a continuation of the trend to develop cold boot attacks for different schemes (as evinced by the literature cited in Chapter 2). But it can also be seen as the beginning of the evaluation of the leading post-quantum candidates against this class of attack. Such an evaluation should form a small but important part of the overall

assessment of schemes in the NIST selection process for post-quantum algorithms.[1] In particular, this chapter evaluates what seems likely to be a leading candidate and lays the groundwork for the later study of other likely candidates in the same broad family of schemes that operate over polynomial rings (such as NTRUprime [7] and various recently proposed ring-LWE-based schemes [3, 12]).

As noted earlier, the exact format in which the private key is stored is critical to developing key recovery attacks in the cold boot setting. This is because the attack depends on physical effects in memory, represented by bit flips in private key bits, and the main input to the attack is a bit-flipped version of the private key. For this reason, it is necessary to either propose natural ways in which keys would be stored in memory in NTRU implementations or to examine specific implementations of NTRU. We adopt the latter approach, and we study two distinct implementations. The first, `ntru-crypto`, is a pair of C and Java libraries developed by OnBoard Security, a spin off of Security Innovation, the patent-holder for some NTRU technology.[2] The second, `tbuktu` is a pair of libraries developed by "Tim Buktu", and is available in 'C' and Java languages.[3] A fork of the Java implementation is included in the popular `Bouncy Castle` Java crypto library.[4]

Each of these implementations stores its private keys in memory in slightly different ways. For example, in Java, `tbuktu` supports a number of different formats, including a representation where the key is stored as 6 lists of indices, each index being a 32-bit integer representing a position where a certain polynomial has a coefficient of value $+1$ or -1. Meanwhile, `ntru-crypto`'s C implementation uses a special representation of polynomial coefficients by trits (three-valued bits), and then packs 5 trits at a time into octets using base-3 arithmetic.

Each of these different private key formats therefore requires a different approach to key recovery in the cold boot setting. In this chapter, we will focus on just a couple of the more interesting cases, where there is some additional structure that we can exploit, or where novel approaches are called for. Nevertheless, we will pose the problem of key recovery in a more general way that makes it possible to see how to generalise our ideas to

[1] See `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` for details of the NIST process.

[2] See `https://github.com/NTRUOpenSourceProject/ntru-crypto` for the code and `https://www.onboardsecurity.com/products/ntru-crypto/ntru-resources` for a list of useful resources related to NTRU.

[3] See `http://tbuktu.github.io/ntru/`.

[4] See `http://bouncycastle.org/`.

cover other cases. Specifically, each of our analyses involves splitting the (noisy) private key into chunks, and creating log-likelihood estimates for each candidate value for each of the chunks. Each such estimate can be regarded as a per chunk score. A log-likelihood estimate (or score) for a candidate for the complete private key can then be computed by summing the per chunk scores across the different chunks. Our problem then becomes one of efficiently enumerating complete candidates and their scores based on lists of candidates for chunks and per-chunk scores, so that each complete candidate can then be tested for correctness (for example, by trial encryption and decryption). It makes sense to perform the enumeration in decreasing order of score if possible, starting with the most likely candidate. This is a problem that also arises in the side-channel attack literature, cf. [69, 11, 48, 47, 18], where, for example, one might obtain scoring information for each byte of an AES key from a power analysis attack and then want to efficiently enumerate and test a large number of complete 16-byte candidates in decreasing order of score until the correct key is found, as was evinced in Chapter 3. We therefore are able to apply the key enumeration algorithms described in Chapter 3 to solve the key recovery problem in our context.

4.2 NTRU Encryption Scheme

In this section we briefly describe the NTRU public key encryption scheme and the NTRU key recovery problem.

Let $N, p, q \in \mathbb{Z}^+$. We define three polynomial rings:

$$R = \mathbb{Z}[x]/(X^N - 1), \quad R_p = \mathbb{Z}_p[x]/(X^N - 1), \quad R_q = \mathbb{Z}_q[x]/(X^N - 1).$$

Thus, for example, elements of R_p can be represented as polynomials of degree at most $N - 1$ with coefficients from $\mathbb{Z}_p$. They can also be represented as vectors of dimension N over $\mathbb{Z}_p$ in the natural way, and we will switch between representations at will.

Definition 4.2.1 *Let $\mathbf{a} \in R_q$ The centred lift of $\mathbf{a}$ to R is the unique polynomial $\mathbf{a}' \in R$ satisfying $\mathbf{a}' \mod q = \mathbf{a}$ whose coefficients all lie in the interval $-\frac{q}{2} < a'_i \leq \frac{q}{2}$.*

Definition 4.2.2 *Let r be a fixed integer and let C_r be the function that, given $\mathbf{a} \in R$, outputs the number of coefficients of $\mathbf{a}$ equal to r. Let $d_1, d_2 \in \mathbb{Z}^+$. We define $T(d_1, d_2) = \{\mathbf{a} \in R \mid C_1(\mathbf{a}) = d_1, C_{-1}(\mathbf{a}) = d_2, C_0(\mathbf{a}) = N - d_1 - d_2\}$. Note that $|T(d_1, d_2)| =$*

$\binom{N}{d_1}\binom{N-d_1}{d_2}$. *An element* $\mathbf{a} \in R$ *is called a ternary polynomial if and only if* $\mathbf{a} \in T(d_1, d_2)$ *for some* $d_1, d_2 \in \mathbb{Z}^+$.

4.2.1 NTRU Public Key Encryption Scheme

The NTRU public key encryption scheme is a lattice-based alternative to RSA and ECC with security that is (informally) based on the problem of finding the shortest vector in a particular class of lattices. The scheme exists in several versions, offering different forms of security (IND-CPA, IND-CCA). The details of the scheme's operation matter less to us than the format of private keys in implementations. However, for completeness, we give an overview of NTRU. We follow the description in [31].

The scheme relies on public parameters (N, p, q, d) with N and p prime, $gcd(p, q) = gcd(N, q) = 1$ and $q > (6d + 1)p$.

Key generation:

1. Choose $\mathbf{f} \in T(d + 1, d)$ that is invertible in R_q and R_p.
2. Choose $\mathbf{g} \in T(d_1, d_2)$ for some $d_1, d_2 \in \mathbb{Z}^+$.
3. Compute $\mathbf{f_p}$, the inverse of $\mathbf{f}$ in R_p.
4. Compute $\mathbf{f_q}$, the inverse of $\mathbf{f}$ in R_q.
5. The public key is $\mathbf{h} = p\mathbf{f_q} \cdot \mathbf{g} \in R_q$; the private key is the pair $(\mathbf{f}, \mathbf{f_p})$.

Encryption On input message $\mathbf{m}$, which we assume to be a centre-lifted version of an element of R_p, and public key $\mathbf{h}$:

1. Choose a random $\mathbf{r}$ with small coefficients, in particular $\mathbf{r}$ can be chosen such that $\mathbf{r} \in T(d, d)$.
2. Compute the ciphertext $\mathbf{e}$ as $\mathbf{e} = \mathbf{r} \cdot \mathbf{h} + \mathbf{m} \in R_q$.

Decryption On input ciphertext $\mathbf{e}$ and private key $(\mathbf{f}, \mathbf{f_p})$:

1. Compute $\mathbf{c} = \mathbf{f} \cdot \mathbf{e}$ in R_q. (Note that this yields $\mathbf{c} = p\mathbf{g} \cdot \mathbf{r} + \mathbf{f} \cdot \mathbf{m}$ over R_q.)
2. Centre-lift $\mathbf{c}$ modulo q to obtain $\mathbf{a}$, and then compute $\mathbf{f_p} \cdot \mathbf{a} \in R_p$. Centre-lift the

result modulo p to obtain $\mathbf{m}$.

We omit the correctness proof for this description of the NTRU scheme. Note that $\mathbf{f_p}$ can be computed from $\mathbf{f}$) on the fly, and so some implementations may only store $\mathbf{f}$ as the private key.

4.2.2 The NTRU Key Recovery Problem

Given $\mathbf{h}$, the task is to find two ternary polynomials, $\mathbf{f}$, $\mathbf{g}$, satisfying $\mathbf{f} \cdot \mathbf{h} = \mathbf{g} \bmod q$. First note that this problem has no unique solution, because if $(\mathbf{f}, \mathbf{g})$ is a solution, then $(x^k \cdot \mathbf{f}, x^k \cdot \mathbf{g})$ is also a solution for all $0 \leq k < N$. The polynomial $x^k \cdot \mathbf{f}$ is called a rotation of $\mathbf{f}$. A rotation $x^k \cdot \mathbf{f}$ is private key in the sense that yields the rotated plaintext $x^k \cdot \mathbf{m}$

If an attacker found a ternary polynomial $\mathbf{f'}$, then $\mathbf{f'}$ would be probably either $\mathbf{f}$ or a rotation of $\mathbf{f}$. To see this, we ask for the probability that some random $\mathbf{f} \in T(d+1, d)$ has the property that $\mathbf{f} \cdot \mathbf{h}$ is a ternary polynomial. Treating the coefficients of $\mathbf{f} \cdot \mathbf{h}$ as independent random variables that are uniformly distributed modulo q, the probability that any particular coefficient is ternary is $\frac{3}{q}$, and hence the probability that every coefficient is ternary is approximately $(\frac{3}{q})^N$. Hence, the expected number of decryption keys in $T(d+1, d)$ is approximately equal to

$$Pr\big(\mathbf{f} \in T(d+1,d) \text{ is a decryption key}\big) \cdot |T(d+1,d)| = (\frac{3}{q})^N \binom{N}{d+1} \binom{N-d-1}{d}.$$

For example, for the standardised parameter set `ees449ep1` targeting 128 bits of security, we have $N = 449$, `d` $= 134$, $q = 2048$, hence the expected number of decryption keys is $(\frac{3}{2048})^{449} \binom{449}{135} \binom{304}{134} \approx 9.856 \times 10^{-1064}$. Therefore, it can be seen from previous calculation that it is unlikely that there are any additional decryption keys beyond $\mathbf{f}$ and its rotations.

4.3 Private Key Formats for NTRU Implementations

In this section we explore the various private key formats in the two implementations we will be working with.

4.3.1 The `tbuktu/Bouncy Castle` Java Implementation.

In this implementation, there are four pieces of information that determine how the private key is stored:

1. A variable `t` that points to a polynomial and from which variables corresponding to the private key components $\mathbf{f}$ and $\mathbf{f_p}$ are constructed.
2. A variable `polyType` that indicates the type of polynomial to use. This can hold two values: `SIMPLE` or `PRODUCT`.
3. A boolean `sparse` that indicates if `t` is a sparse or dense polynomial. This variable applies only if `polyType` has value `SIMPLE`.
4. A boolean `fastFp` that indicates the manner in which $\mathbf{f}$ is built from `t`. If `fastFp` = `true`, then $p = 3$, $\mathbf{f} = 1 + 3\mathtt{t}$ and $\mathbf{f_p} = 1$; otherwise $\mathbf{f} = \mathtt{t}$ and $\mathbf{f_p} = \mathtt{t}^{-1} \bmod p$. This relates to an implementation trick for the case $p = 3$.

When `polyType` has the value `SIMPLE`, `t` will be either a dense ternary polynomial or a sparse ternary polynomial, as determined by the value of `sparse`. In the dense case, `t` is represented as an `int` array of length N whose entries have values from $\{-1, 0, 1\}$. In memory, each entry is stored as a 32-bit signed integer, using two's complement, i.e. $+1$ is stored as the 32-bit string $000\ldots01$, 0 is stored as $000\ldots00$ and -1 is stored as $111\ldots11$. Meanwhile, in the sparse case, `t` is represented as two `int` arrays, `ones` and `negOnes`, where:

1. The array `ones` contains the indices of the $+1$ coefficients of `t` in increasing order (so that the entries in the area are 32-bit representations of integers in the range $[0, N-1]$).
2. The array `negOnes` contains the indices of the -1 coefficients of `t` in increasing order (with entries having the same bit representation as the entries of `ones`).

When `polyType` has the value `PRODUCT`, `t` will be a product form polynomial. In this case, `t` is represented by three different sparse ternary polynomials $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$ such that $\mathbf{t} = \mathbf{f}_1\mathbf{f}_2 + \mathbf{f}_3$. All three of $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$ are stored in memory separately in sparse form. This means that, when `polyType` has the value `PRODUCT`, then the private key is represented in memory by a total of 6 `int` arrays $\mathbf{f}_i$.`ones`, $\mathbf{f}_i$.`negOnes`, $1 \le i \le 3$.

Note that the private key formats for the `tbuktu` C implementation are largely the same as for the Java one, and so we do not detail them further here.

4.3.2 Reference Parameters for `tbuktu`

The `tbuktu` implementation includes 10 named reference parameter sets with a range of choices for N and q, targeting different security levels and optimisations. These 10 sets are detailed in the `EncryptionParameters` class.

For example, both `APR2011_439` and `APR2011_439_FAST` parameter sets target 128 bits of security. Both are included in the standardised parameter sets from IEEE 1363.1-2008 [75]. If the former set is selected, $\mathbf{f} = \mathbf{t}$, with $\mathbf{t}$ being represented as a sparse ternary polynomial with $\texttt{df} = 146$ coefficients set to $+1$ and with $\texttt{df} - 1$ of them set to -1. If the latter set is selected, then $\mathbf{f} = \mathbf{1} + 3\mathbf{t}$, with $\mathbf{t} = \mathbf{f}_1\mathbf{f}_2 + \mathbf{f}_3$; moreover $\mathbf{f}_1$ has $\texttt{df1} = 9$ coefficients set to each of $+1$ and -1, so $\mathbf{f}_1 \in T(9, 9)$ (while $\texttt{df2} = 8$ and $\texttt{df3} = 5$ for $\mathbf{f}_2$ and $\mathbf{f}_3$, respectively). In [34], security analyses are provided for the standardised product-form parameter sets from IEEE 1363.1-2008.

4.3.3 The `ntru-crypto` Java Implementation

Here, the private key $\mathbf{f}$ is always of the form $\mathbf{1} + 3\mathbf{t}$ where $\mathbf{t}$ is a ternary polynomial, and we have $p = 3$ (so that $\mathbf{f_p} = 1$). In this implementation, $\mathbf{f}$ is stored directly in memory as an array of short integers. That is, the coefficients $f_0, f_1, \ldots, f_{N-1}$ of $\mathbf{f}$ are stored as a sequence of 16-bit signed two's complement integers with $f_0 \in \{-2, 1, 4\}$ and $f_i \in \{-3, 0, 3\}$, for $1 \leq i < N$.

4.3.4 The `ntru-crypto` C Implementation

In this particular implementation, `ntru_crypto_ntru_encrypt_keygen` routine generates a key pair. It generates $\mathbf{f}$ as either a product of polynomials ($\mathbf{f}_1 \cdot \mathbf{f}_2 + \mathbf{f}_3$) or a single polynomial. To represent $\mathbf{f}$, a list, index-based representation, consisting of the concatenation of two sublists: the sublist containing the indices of the $+1$ coefficients and the sublist containing the indices of the -1 coefficients. Each entry is stored in an unsigned 16 bit integer. This list is then used to construct a bit string following the exact format shown by Table 4.1.

tag	count	OID	OID	packed pubkey [packed privkey]

Table 4.1: Format of private key blob in `ntru-crypto` C Implementation

In this bit string, byte 0 is a tag to represent the format being used to pack the private key; byte 1 represents the number of bytes used for OID; bytes 2-4 hold the OID, which is an identifier to get the parameter set. From byte 5 onwards, there is the packed public key followed by the packed private key.

The private key is packed according to a parameter that takes two possible values:

I. The value `NTRU_ENCRYPT_KEY_PACKED_TRITS`: In this case $\mathbf{f}$ is a single, non-sparse polynomial and so $\mathbf{f}$ is packed by converting $\mathbf{f}$'s index-based representation into a coefficient-based representation, which is packed. Specifically, creating an array by setting $+1$ at each index for which $\mathbf{f}$ has coefficient $+1$, setting 2 at each index for which $\mathbf{f}$ has coefficient -1, and 0 otherwise. This array is packed by transforming blocks of 5 consecutive values into a single value, treating a block as a base-3 number. This new value is stored in an octet.

II. The value `NTRU_ENCRYPT_KEY_PACKED_INDICES`: In this case $\mathbf{f}$ is either in product form or a single sparse polynomial. For each entry of $\mathbf{f}$'s index-based representation, its value is extracted[5] and appended to form a long bit string. This bit string is then divided into octets.

4.3.5 Reference Parameters for `ntru-crypto`

The `ntru-crypto` implementation includes 12 named reference parameter sets with a range of choices for N and q, targeting different security levels and optimisations. For the Java implementation, these parameter sets are defined in the `KeyParams` class. The values of N range from 401 to 1499, with $p = 3$ and $q = 2048$ throughout; the number of $+1$'s and -1's in $\mathbf{f}$ (resp. $\mathbf{g}$), denoted `df` (resp. `dg`) depends on N; for example, for the parameter set `ees449ep1` targeting 128 bits of security, we have $N = 449$, `df` $= 134$, and `dg` $= 149$. Moreover, for the parameter set `ees677ep1` targeting 192 bits of security, we have $N = 677$, `df` $= 157$, `dg` $= 225$. Both parameter sets are included in the standardised parameter sets from IEEE 1363.1-2008 [75]. Furthermore, security analyses are provided

[5] only the relevant bits out of the 16 bit string.

for various parameter sets in [9].

4.4 Mounting Cold Boot Key Recovery Attacks

In this section we present our cold boot key recovery attacks on the implementations and corresponding private key formats introduced in the previous section. First, we will describe the general key recovery algorithm and secondly we consider the `ntru-crypto` implementations. We will then consider the `tbuktu` Java implementation in which the `PRODUCT` form of private key is used and such that `fastFp` = `true`, so that $\mathbf{f} = \mathbf{1} + 3\mathbf{t}$ with $\mathbf{t} = \mathbf{f}_1\mathbf{f}_2 + \mathbf{f}_3$ where all three of $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$ are stored in memory in sparse form.

We continue to make the assumptions outlined in Section 2.3, and additionally assume that all relevant public parameters and private key formatting information are known to the adversary.

4.4.1 Key Recovery Strategy

Let $\mathtt{r} = (b_0, \ldots, b_{W-1})$ denote the bits of the noisy encoding of the key and let us name the chunks $\mathtt{r}^0, \mathtt{r}^1, \ldots, \mathtt{r}^{\mathcal{N}-1}$ so that $\mathtt{r}^i = b_{i\cdot w} b_{i\cdot w+1} \ldots b_{i\cdot w+(w-1)}$. Furthermore, the attacker represents $\mathtt{r}$ as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j}, with $n_{b^j} > 0$, consecutive chunks, such that $\mathcal{N} = \sum_{j=0}^{n_b-1} n_{b^j}$. Therefore,

$$\mathtt{r} = \mathtt{b}^0 || \mathtt{b}^1 || \cdots || \mathtt{b}^{n_b-1},$$

where

$$\mathtt{b}^j = \mathtt{r}^{i_j} || \mathtt{r}^{i_j+1} || \cdots || \mathtt{r}^{i_j+n_{bj}-1},$$

for $0 \leq j < n_b$ and some $0 \leq i_j < \mathcal{N}$. The attacker now proceeds as follows.

`Phase I`

- For each chunk $\mathtt{r}^i$, with $0 \leq i < \mathcal{N}$, the attacker uses Equation (2.1) to compute a log-likelihood score for each candidate $\mathtt{c}^i$ for the chunk $\mathtt{r}^i$, viz.

$$\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i] = n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta),$$

where the n_{ab}^i values count occurrences of bits across the i-th chunks, $\mathtt{r}^i$, $\mathtt{c}^i$. Therefore, the attacker produces a list containing up to 2^w chunk candidates, L_{r^i}.

- For each block $\mathtt{b}^j$, the attacker presents the lists $L_{r^{i_j}}, \ldots, L_{r^{i_j+n_{b^j}-1}}$ as inputs to an instance of OKEA, which was described in Section 3.2.1, to produce a list of the M_j highest scoring candidates for the block $\mathtt{b}^j$, L_{b^j}.

Once `Phase I` completes, the attacker then proceeds as follows.

`Phase II`

- The lists L_{b^j} are given as inputs to an instance of a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block $\mathtt{b}^j$. This instance will generate high scoring candidates for the encoding of the key. For each complete candidate $\mathtt{r}'$ output by the enumeration algorithm, it is given as input to a verification function V to verify whether the candidate is valid or not. This function is case-dependent.

In `Phase II`, non-optimal key enumeration algorithms would suit better, since they are parallelisable and memory-efficient. Even though their outputs may not be given in the optimal order, the search can be customisable by either selecting a suitable interval in which the outputs' scores lie, e.g. the algorithm described in Section 3.2.3, or selecting a suitable interval in which the outputs' ranks lie, e.g. the algorithm described in Section 3.2.5.

4.4.2 The `ntru-crypto` Java Implementation

Recall from Section 4.3.3 that the coefficients of $\mathbf{f}$ are stored directly in memory as an array of 16-bit, signed two's complement integers, with $f_0 \in \{-2, 1, 4\}$ and $f_i \in \{-3, 0, 3\}$, for $1 \leq i < N$. For simplicity, we assume that f_0 is known (there are only 3 possible values for f_0 and the attack can be repeated for each possible value). The attacker then receives a noisy version $\mathtt{r} = (\mathtt{r}_0, \ldots, \mathtt{r}_{W-1})$ of the array with entries $f_1, \ldots, f_{N-1}$ which is $W = 16(N-1)$ bits in size. In the terminology of Section 4.4.1, we set $w = 16$ and partition the noisy key into $W/w = N-1$ chunks $\mathtt{r}^i$, each chunk corresponding to a single, 16-bit encoded coefficient f_{i+1}.

To apply the key recovery strategy in this case, we simply set any block to have only a chunk, i.e., $\mathtt{b}^i = \mathtt{r}^i$. The first step of `Phase I` then uses Equation (2.1) to compute log-likelihood scores $\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i]$ for each chunk i and each candidate $\mathtt{c}^i$ for the chunk. Note that for each chunk i, there are only 3 possible candidates $\mathtt{c}^i$, since $f_i \in \{-3, 0, 3\}$ (in the general formulation with $w = 16$ there could be up to 2^{16} candidates per chunk). Therefore, we obtain $N - 1$ lists of chunk candidates, each list containing 3 entries. The second step of `Phase I` is then sort these lists in decreasing order based on their score component.

The task of `Phase II` is to combine chunk candidates, one per chunk, to generate complete private key candidates `c` with high scores, which can then be tested via trial encryption and decryption. In order to generate complete private key candidates `c` with high scores, we employ an algorithm that is closely based on that of [48]. Specifically, we use a variant of the non-optimal key enumeration algorithm introduced in Section 3.2.3. This employs a stack, with partial cumulative scores for candidates at "depth" i in the search being computed by adding the chunk score at depth i to a cumulative score for the candidates at "depth" $i-1$. Once "depth" $N-1$ is reached, and a complete candidate is generated, the candidate can be filtered and then tested (in fact, the known restrictions on the number of $+3$ and -3 coefficients in private keys for the standard parameters that we are attacking can be used to perform early aborts on partial candidates). As described in Section 3.2.3, we can restrict the search space to certain intervals of scores by appropriate pruning of partial solutions and make use of parallelisation.

4.4.3 The `ntru-crypto` C Implementation

Now we turn our attention to the `ntru-crypto` C implementation. Recall from Section 4.3.4 that there are two cases.

For the case I, the coefficients of $\mathbf{f}$ are stored in memory as an array of 8-bit unsigned integers, where a single entry represents a block of 5 consecutive coefficients (they are treated as a base-3 number). Let us suppose that the size of this array is N_r. Hence, the attacker receives a noisy version $\mathtt{r} = (\mathtt{r}_0, \ldots, \mathtt{r}_{W-1})$ of the array of $W = 8 \cdot N_r$ bits.

To apply the key recovery strategy in this case, the attacker simply sets each chunk $\mathtt{r}^i$ to be an 8-bit string and sets a block to have only a single chunk, i.e. $\mathtt{b}^i = \mathtt{r}^i$. Note that

the candidates for any chunk are the integers in the interval $[0, 242]$, since an entry in the original encoding represents a 5-digit base-3 number and its range of values goes from $(00000)_3 = 0_{10}$ to $(22222)_3 = 242_{10}$. The attacker then uses Equation (2.1) to compute log-likelihood scores $\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i]$ for each chunk i and each candidate $\mathtt{c}^i$ for that chunk. Therefore, the attacker constructs N_r lists of chunk candidates, each list containing 243 entries. The attacker then sort these lists in decreasing order based on their score component (second step of `Phase I`).

Concerning `Phase II`, the task is then to combine chunk candidates, one per chunk, to generate complete, high-scoring private key candidates `c`, which can then be tested via a verification function. For this task, the attacker may employ a non-optimal key enumeration algorithm, e.g. the algorithm of [48] described in Section 3.2.3. Regarding the verification function, this simply unpacks the output candidate `c` and constructs a polynomial $\mathbf{f}'$ from it. It then verifies if $\mathbf{f}'$ is a ternary polynomial by counting the number of $+1$ coefficients and -1 coefficients. If so, a trial encryption/decryption with $\mathbf{f}'$ as private key may be performed for correctness.

Regarding the case II, the coefficients of $\mathbf{f}$ are stored in memory as an array of 8-bit unsigned integers, representing a sequence of indices. Let us suppose that this size of this array is N_r. Hence, the attacker receives a noisy version $\mathtt{r} = (\mathtt{r}_0, \ldots, \mathtt{r}_{W-1})$ of the array of $W = 8 \cdot N_r$ bits.

To apply the key recovery strategy in this case, the attacker simply sets a chunk $\mathtt{r}^i$ to be an 8-bit string and sets a block to have only a chunk, i.e. $\mathtt{b}^i = \mathtt{r}^i$. Note that the candidates for any chunk are the integers in the interval $[0, 255]$. The attacker therefore uses Equation (2.1) to compute log-likelihood scores $\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i]$ for each chunk i and each candidate $\mathtt{c}^i$ for that chunk. So the attacker construct N_r lists of chunk candidates, each list containing 256 entries. Concerning `Phase II`, the attacker may employ a non-optimal key enumeration algorithm, e.g. the algorithm of [48] described in Section 3.2.3. The verification function V simply unpacks the complete array candidate and constructs a polynomial $\mathbf{f}'$ from it. $\mathbf{f}'$ can be either a single, sparse polynomial or of the form $\mathbf{f}_1 \cdot \mathbf{f}_2 + \mathbf{f}_3$. A trial encryption/decryption with $\mathbf{f}'$ as private key may be performed for correctness.

4.4.4 The `tbuktu` Java Implementation

Now we turn our attention to the `tbuktu` Java implementation in some of its more interesting cases. Recall from Section 4.3.1 that when the `PRODUCT` form of private key is used and when `fastFp = true`, then we have $\mathbf{f} = \mathbf{1} + 3\mathbf{t}$ with $\mathbf{t} = \mathbf{f}_1\mathbf{f}_2 + \mathbf{f}_3$ where all three of $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$ are stored in memory in sparse form.

This means that we have 6 arrays of indices in memory $\mathbf{f}_i$.`ones`, $\mathbf{f}_i$.`negOnes`, $1 \leq i \leq 3$. Each array is of type `int` and each entry in each array stores the position of either a $+1$ or a -1 coefficient in one of the polynomials $\mathbf{f}_i$; moreover the entries should be in increasing order. We assume the starting positions in memory, total sizes, and ranges of possible values in each of these tables is known. We also know that for any pair $\mathbf{f}_i$.`ones`, $\mathbf{f}_i$.`negOnes`, the two tables of values should be non-intersecting. We let S_i denote the common length of the two arrays $\mathbf{f}_i$.`ones`, $\mathbf{f}_i$.`negOnes` (this is determined by the parameters used to generate the private key).

We now present the two-phase attack to generate complete private key candidates.

Regarding `Phase I`, we apply a modified version of the optimal key enumeration algorithm introduced in Section 3.2.1. As in the description in Section 3.2.1, this algorithm takes as input a collection of W/w lists of candidates, one list per chunk, and produces as output a list of `lsize` complete candidates, each across all W bits. It uses a dynamic programming version of a list merging strategy to generate complete candidates in decreasing order of score. OKEA has the property that it is guaranteed to output the `lsize` highest scoring (i.e. most likely) candidates across all the chunks (hence its optimality). It seems to be particularly effective when W/w, the number of chunks being considered, is moderate (see Section 3.3); additionally it was applied in the case of reconstructing 16-byte AES keys from their bytes, with 16 chunks [69].

We perform this step for each of our 6 arrays as follows: we build W/w lists of candidates, setting $w = 32$ and $W = wS_i$ so that we have S_i chunks. Each chunk corresponds to one `int` entry in the array, and each list is of size N (since, at the outset, every chunk could take on any value between 0 and $N - 1$, these being the possible indices of a $+1$ or -1 coefficient). The score for each entry in each list is obtained using our per-chunk log-likelihood expression (Equation (2.1)). We modify the OKEA algorithm in such a way that

it is guaranteed to output the top `lsize` candidates by score which additionally respect our ordering requirement – that is, the entries in a candidate should be in increasing order of size. This modification is done by adding an extra filtering step in each merge phase of OKEA which removes candidates that do not respect the ordering constraint.

At the end of this step, then, we obtain 6 lists L_{b^i}, $1 \leq i \leq 6$, the entries of each list comprising `lsize` high-scoring candidates for one of the 6 arrays $\mathbf{f}_i$.`ones`, $\mathbf{f}_i$.`negOnes`, $1 \leq i \leq 3$.

Concerning the second phase of the attack, we present these 6 lists as inputs to a non-optimal key enumeration algorithm as described in Section 4.4.1 above. In particular, we perform a stack-based, depth-first search on the lists, regarding each list as giving a set of candidates on one of 6 chunks. Each complete candidate (on 6 chunks) now gives a candidate for $\mathbf{f}_i$.`ones`, $\mathbf{f}_i$.`negOnes`, $1 \leq i \leq 3$, these being tables of the indices where the component polynomials $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$ have coefficients $+1$ and -1. We then apply the constraint that the pairs of tables be non-intersecting (applying it earlier in the process is not very efficient, since the probability of a collision of indices is small for the parameters of interest). If a candidate survives this filter, we can construct the full private key $\mathbf{f} = \mathbf{1} + 3\mathbf{t}$ with $\mathbf{t} = \mathbf{f}_1 \mathbf{f}_2 + \mathbf{f}_3$ and test it for correctness.

As before, `Phase II` is amenable to parallelisation and to searching over restricted score intervals. Now the parameters are more akin to those studied in the prior work [48, 47] – we have 6 chunks, and `lsize` candidates per chunk, with typical values for `lsize` in our experiments being 256, 512 and 1024.

4.5 Experimental Evaluation

4.5.1 Implementation

All the algorithms discussed in this chapter were implemented in Java. We choose Java for several reasons. First, the two implementations that we have studied in this chapter were written in Java (as well as C). Second, our key recovery algorithm makes use of some of the implemented key enumeration algorithms described in Chapter 3. Additionally, Java platform provides classes and interfaces implementing commonly reusable collection data structures and supports concurrent programming.

4.5.2 Parallelisation

We made extensive use of parallelisation in our implementations, particularly for the stack-based, depth-first search that is at the core of the attacks. We made use of the combination of the two parallelisation methods described in Section 3.2.3.5. The first parallelisation method involves splitting up the range of scores of interest into n disjoint, equal-sized sub-intervals. The second method involves splitting the list of candidates for the first chunk in our algorithm into m equal-sized sub-lists, and running the algorithm as a separate task for each sub-list, thereby constraining solutions from each task to begin with a chunk from the specified sublist for that task. These two approaches are combined to execute mn threads in parallel. Of course, as soon as one of the threads completes and successfully finds the private key, the others can all be aborted.

4.5.3 Search Intervals

Defining appropriate search intervals on which to run our algorithms is important in guaranteeing the success of our attacks within a reasonable amount of running time. Recall that, given a collection of lists as input, each list containing candidate for chunks and their scores, our algorithms will consider all possible candidates with total scores in any specified interval $[a, b]$. We considered two distinct classes of search interval:

1. Class I intervals are the form $[\mu - W, \mu + W]$, where μ is the average score of the correct key and W is some real number that is tuned to the maximum running time available. Here μ can be computed empirically by generating many private keys, flipping their bits according to the error probabilities α, β, and then using the usual log-likelihood scoring function. Using such intervals capture the intuition that it might be better to examine key candidates that are situated around the average score, since these are more likely to be correct. This of course violates the principle of the maximum likelihood approach.
2. Class II intervals are of the form $[\max - W, \max]$, where max is the maximum possible score and W is again a real number that can be tuned. Here, the value of max is easily calculated by summing across the highest scoring entries in each list. Searching in such intervals better matches the approach of maximum likelihood estimation.

4.5.4 Simulations

To simulate the performance of our algorithms, we generate a private key (according to some chosen format), flip its bits according to the error probabilities α, β, and then run our chosen algorithm with selected parallelisation parameters m, n and interval definition $[a, b]$. We refer to such a run attempting to recover a single private key as a *simulation*.

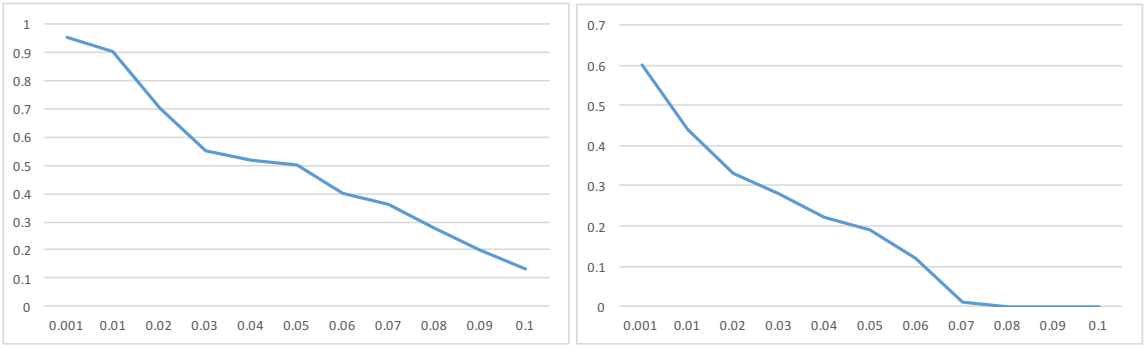
For our experiments, we ran our simulations on a machine with Intel Xeon CPU E5-2667 v2 cores running at 3.30GHz; we used up to 16 cores. In order to run our simulations concurrently, a pool of threads is initialised with a maximum number of threads given as a parameter. When a simulation is to be run and tested, it generates its various tasks according to the given parameters, each of which then is submitted to the main pool in order. After it has finished, a thread outputs either the recovered private key or null value (indicating failure to find the key) along with some statistics. Note that having a pool created with a defined number of threads helps to avoid exhausting and reusing computational resources, in contrast to creating a new thread per task.

4.5.5 Results for the `ntru-crypto` Java Implementation

Here, we only considered Class II intervals, i.e. intervals of the form $[\max - W, \max]$. To calculate suitable values for W, we used random sampling from the set of possible candidates (by choosing chunks at random from each list) in order to estimate σ, the standard deviation of the candidate scores. We then set W as $r\sigma$ and experimented with different values of r, the idea being that larger values of r would correspond to bigger intervals, including more candidates and giving a higher chance of success at the cost of more computation. We used 2^{20} candidates in sampling to estimate σ.

After manual tuning, the number of tasks was set to 3, r was set to 0.01 and the number of subintervals m was set to 1. Hence, in our experiments, searches were conducted over the interval $[\max - 0.01\sigma, \max]$ with 3 tasks.

Figure 4.1a shows the success rate of our attack for the `ees449ep1` parameters ($N = 449$, `df` $= 134$, `dg` $= 149$, $p = 3$, and $q = 2048$). Figure 4.1b shows the success rate for the `ees677ep1` parameters($N = 677$, `df` $= 157$, `dg` $= 225$, $p = 3$, and $q = 2048$).



(a) `ees449ep1` parameters. (b) `ees667ep1` parameters.

Figure 4.1: Success rate of our algorithm (y-axis) against β (x-axis) for a fixed $\alpha = 0.001$, using Class II intervals.

It can be seen from the two figures that the success rate is acceptably high for small values of β, but rapidly reduces as β is increased in size. Increasing the size of r (and therefore the search interval $[\max - r\sigma, \max]$) would improve the success rate at the cost of increased running time. For $r = 0.01$, we saw running times on the order of minutes to hours. There were a few simulations with very high running times; these were aborted after 1 day of computation. We observed this behaviour in particular for high values of β. In this case, the number of tasks, 3, the number of chunks, 400, and the nature itself of what was considered a suitable candidate (number of 1's and -1's) made it hard to predict the number of candidates in a given interval. So searching over a given interval is done somewhat "blindly", in the sense that searching over the interval $[\max - 0.01\sigma, \max]$ will not behave in a consistent manner in terms of the number of candidates found (and hence the running time needed).

4.5.6 Results for the `tbuktu` Java Implementation

Due to the additional structure of private keys compared to the `ntru-crypto` implementation, we focussed a greater experimental effort on the `tbuktu` Java implementation.

4.5.6.1 Counting Candidates and Estimating Running Times

Because of the nature of the log-likelihood function employed to calculate scores, each of the six lists L_{b^i} output by `Phase I` of the attack will have many repeated score values. This enables us to efficiently compute the *number* of candidates that `Phase II` of the attack will consider in any given interval $[a, b]$. To do this, we run a modified version of `Phase`

`II` in which the lists L_{b^i} are replaced by "reduced" lists which eliminate chunk candidates having repeated score values, and include the counts (numbers) of such candidates along with their common score. By simultaneously computing the sums of scores and *products* of counts on these reduced lists, we can compute the total number of candidates that will have a given score, over all possible scores in any chosen interval.

Because the size of each reduced list is less than 10 on average in our experiments (when `lisze` is up to 1024), we obtain a very efficient algorithm for counting the number of candidates in any given interval that our `Phase II` search algorithm would need to consider. We can combine this counting algorithm with the average time needed to generate and consider each candidate to get estimates for the total running time that our algorithm would encounter for a given choice of interval. We can then also compute the expected success probability and estimated running time (for the given number of candidates or given interval considered) without actually running the full `Phase II` search algorithm.

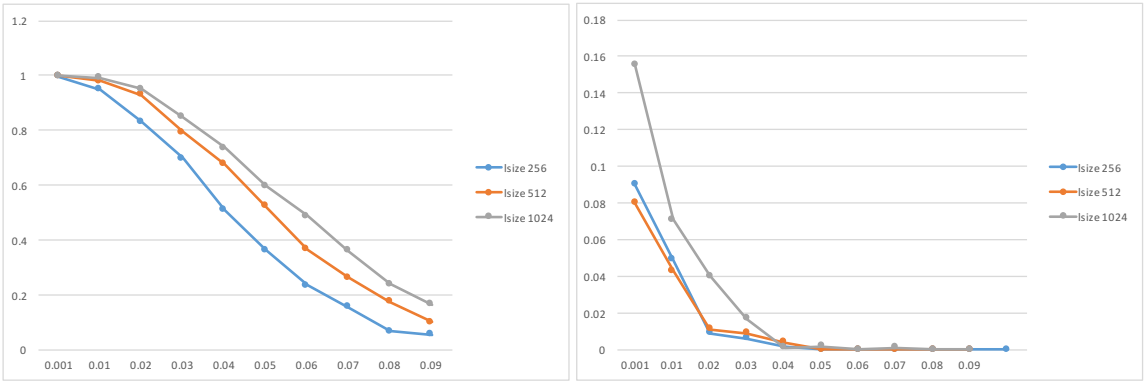
Note that this counting algorithm may be seen as a very special case of Algorithm 13 described in Section 3.2.5. In the sense that we may construct a "standard histogram" H_i for each L_{b^i} such that a bin of H_i contains all chunk candidates of L_{b^i} with a common repeated score. Then these "standard histograms" are given as inputs to Algorithm 13 and so the resulting "convoluted histogram" $H_{0:5}$ allows us to both efficiently count the number of candidates whose scores lie in a given interval and efficiently construct an interval of some given form such that the number of candidates whose scores lie in the interval is at least a given value but the numerical difference with such given value may be minimised.

4.5.6.2 Parameters

The encryption parameters used for running the simulations are `APR2011_439_FAST` ($N = 439$, $p = 3$, $q = 2048$, `df1` $= 9$, `df2` $= 8$, `df3` $= 5$, `sparse` $=$ `true`, `fastP` $=$ `true` so that $\mathbf{t} = \mathbf{f}_1\mathbf{f}_2 + \mathbf{f}_3$, and $\mathbf{f}_1 \in T(9,9)$, $\mathbf{f}_2 \in T(8,8)$, $\mathbf{f}_3 \in T(5,5)$).

4.5.6.3 Results – Complete Enumeration

In our experiments, we set `lsize` to 2^r for `Phase I`, for $r = 8, 9, 10$. Thus, six candidate lists each of size 2^r will be obtained from `Phase I`. Let p_i denote the probability that the correct candidate is actually found in the i-list; p_i will be a function of r. It follows



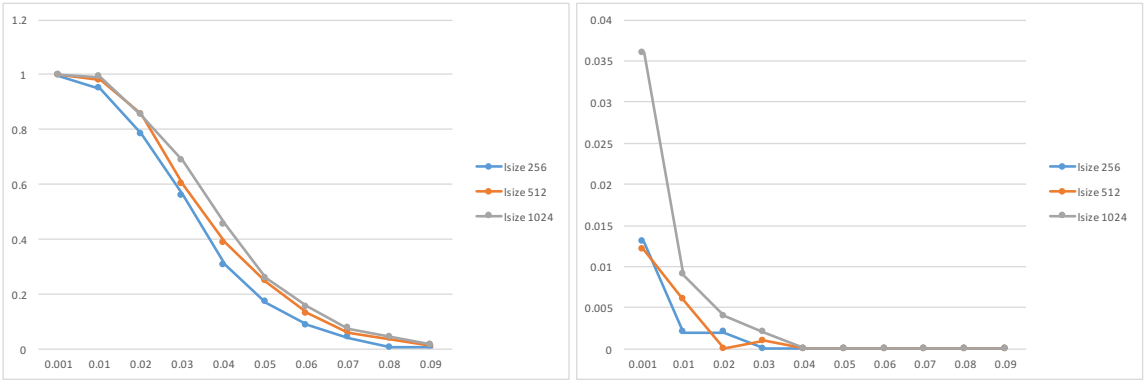
(a) $\alpha = 0.001$. (b) $\alpha = 0.09$

Figure 4.2: Expected success rate for a full enumeration for $\alpha = 0.001, 0.09$. The y-axis represents the success rate, while the x-axis represents β.

that the probability that our `Phase II` algorithm outputs the correct private key when performing a *complete* enumeration over all 2^{6r} candidate keys is given by $p = \prod_{i=1}^{6} p_i$. This simple calculation gives us a way to perform simulations to estimate the expected success rate of our overall algorithm (`Phase I` and `Phase II`) without actually executing the expensive `Phase II`. We simply run many simulations of `Phase I` for the given value of `lsize` (each simulation generating a fresh private key and perturbing it according to α, β), and, after each simulation, test whether the correct chunks of the private keys are to be found in the lists.

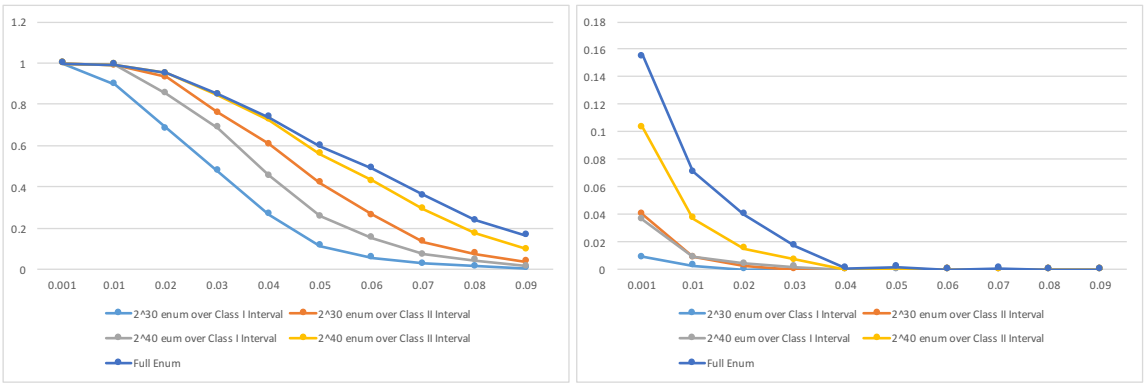
Figure 4.2 shows the success rates for complete enumeration for values of `lsize` $= 2^r$ for $r \in \{8, 9, 10\}$. As expected, the greater the value of `lsize`, the higher the success rate for a fixed α and β. Also, note that when the noise is high (for example $\alpha = 0.09$ and $\beta = 0.09$), the success rate drops to zero. This is expected since it is likely that at least one chunk of the private key will not be included in the corresponding list coming out of `Phase I` when the noise levels are high, at which point `Phase II` inevitably fails.

Note that each data point in this figure (and all figures in section) were obtained using 100 simulations. Note that the running times for `Phase I` are very low on average (≤ 50 ms), since that phase consists of calling the OKEA for each of the six lists with `lsize` in the set $\{256, 512, 1024\}$.



(a) $\alpha = 0.001$. (b) $\alpha = 0.09$

Figure 4.3: Success rate for enumeration with 2^{40} keys over a Class I interval for $\alpha = 0.001, 0.09$, for different values of `lsize`. The y-axis represents the success rate, while the x-axis represents β.



(a) $\alpha = 0.001$. (b) $\alpha = 0.09$

Figure 4.4: Success rates for full enumeration, and partial enumeration with 2^{30} keys, 2^{40} keys for $\alpha = 0.001, 0.09$ and with `lsize` $= 1024$. The y-axis represents the success rate, while the x-axis represents β.

4.5.6.4 Results – Partial Enumeration

Here, we exploit our counting algorithm to estimate success rates as a function of the total number of keys considered, K. Specifically, given a value K, and an interval type (I or II), we can set W accordingly so that the right number of keys will be considered. Since we can easily estimate the speed at which individual keys can be assessed, we can also use this approach to control the total running time of our algorithms.

Figure 4.3 shows how the success rate of our algorithm varies for different values of `lsize`, focusing on Class I intervals. We observe the same trends as for full enumeration, i.e. the greater is `lsize`, the higher is the success rate for a fixed α and β. Also, for larger values of (α, β), the success rate drops rapidly to zero.

Figure 4.4 shows the success rates for a complete enumeration and partial enumerations with 2^{30} keys and 2^{40} keys, for both Class I and Class II intervals. As expected, the success rate for a full enumeration is greater than for the partial enumerations (but note that a full enumeration here would require the testing of up to 2^{60} keys, which may be a prohibitive cost). Note that the closest success rate to the success rate of a full enumeration is achieved with partial enumerations with 2^{40} keys over a Class II interval, and that partial enumerations over Class I intervals perform poorly, in the sense that their success rates are even dominated by the success rate of enumerations with 2^{30} keys over Class II intervals. The superiority of Class II intervals is in-line with the intuition that testing high log-likelihood candidates for correctness is better than examining average log-likelihood ones.

4.5.6.5 Running times

From our experiments, we find that our code is able to test up to 1200 candidates per millisecond per core during `Phase II`. This value may vary in the range 700–1200 when there are multiples tasks running. The reason for this variation may be the cost associated with the Java virtual machine (particularly, its garbage collector). Using only a single core, an enumeration of 2^{30} (2^{40}) candidate keys will take about 14 minutes (10 days, respectively).

4.6 Chapter Conclusions

In this chapter, we have initiated the study of cold boot attacks for the NTRU public key encryption scheme, likely to be an important candidate in NIST's ongoing post-quantum standardisation process. We have proposed algorithms for this problem, with particular emphasis on two existing NTRU implementations and two private key formats. We have experimented with the algorithms to explore their performance for a range of parameters, showing how algorithms developed for enumerating keys in side-channel attacks can be successfully applied to the problem. Our attacks do not exploit the underlying mathematical structure of the NTRU scheme. It would be interesting to explore whether our techniques can be combined with other approaches, such as lattice-reduction, to further improve performance. Even though we focused mainly on the two available Java implementations, we expect to get similar results for the C implementations.

Chapter 5

Cold Boot Attacks on BLISS

Contents

5.1 Introduction 95
5.2 Preliminaries 97
5.3 BLISS Signature Scheme 99
5.4 The strongSwan Project 100
5.5 Mounting Cold Boot Key Recovery Attacks 102
5.6 Experimental Evaluation 118
5.7 Chapter Conclusions 124

In this chapter, we analyse the feasibility of cold boot attacks against the BLISS signature scheme. We first review the in-memory format for the private key of the implementation provided by the ***strongSwan*** *project. We then propose a key recovery strategy based on key enumeration algorithms and then establish a connection between the key recovery problem in this particular case with an instance of Learning with Errors Problem (LWE). We also explore other techniques based on the meet-in-the-middle generic attack to tackle this instance of LWE.*

5.1 Introduction

In this chapter, we examine the feasibility of cold boot attacks against the BLISS signature scheme [22], a member of the broad family of schemes that operate over polynomial rings. We believe this to be the first time that this has been attempted, however there are very recent studies attempting other side channels on this signature scheme [13, 24, 58]. Our work is the continuation of the trend to develop cold boot attacks for different schemes

as revealed by the literature cited in Section 2.2. But it is also the continuation of the evaluation of post-quantum cryptographic schemes against this class of attack. Such an evaluation should form a small but important part of the overall assessment of these schemes. This scheme, however, has not been submitted as candidate to the ongoing NIST standardisation process of post-quantum signature schemes.

As noted previously, when developing key recovery attacks in the cold boot setting, it is important to know the exact format used to store the private key in memory. The reason for this is that this attack depends on the physical effects in memory, which causes bit flips in the binary representation of the private key. Also, the main input to this attack is a bit-flipped version of the private key. Therefore, it is necessary to either propose natural formats in which a private key would be stored in memory or review specific implementations of this scheme. As in the previous chapter, we adopt reviewing existing implementations, and so we study the BLISS implementation provided by the `strongSwan` project, an OpenSource IPsec implementation. This implementation particularly stores its private key in memory in an interesting way therefore requiring novel approaches to key recovery.

In this chapter, we will present various approaches to key recovery. We first analyse the key recovery problem in this particular case via key enumeration, and so propose different techniques for key recovery. Specifically, our analysis involves splitting the components of the private key into chunks, generating high scoring candidates for each chunk and combining them to obtain high scoring candidates for the private key. We achieve this by making use of key enumeration algorithms, which are discussed at length in Chapter 3. In particular, we will make use of the general key recovery strategy developed in Chapter 4 as a core algorithm.

We then turn our attention to exploit further the algebraic relation among the components of the private key, and we thus establish a connection between the key recovery problem in this particular case and an instance of Learning with Errors Problem (LWE). We then explore various key recovery techniques to tackle this instance of LWE, such as the meet-in-the-middle generic attack (collision techniques), as well as key recovery approaches based on lattice techniques. In particular, we show a key recovery strategy combining lattice techniques and key enumeration.

5.2 Preliminaries

In this section, we will introduce the notation and some concepts that we will use in this chapter.

5.2.1 Notation

In this chapter, we write vectors and polynomials in bold lowercase letters, e.g. $\mathbf{a}$, and matrices in bold uppercase letters, e.g. $\mathbf{A}$. We frequently identify polynomials $\mathbf{a} = \sum_{i=0}^{n-1} a_i x^i$ with their coefficient vector $\mathbf{a} = (a_0, \ldots, a_{n-1})$ or $\mathbf{a}^t$. For any integer q, we identify the ring $\mathbb{Z}_q$ with the interval $[-q/2, q/2) \cap \mathbb{Z}$, and in general for a ring R, we define R_q to be the quotient ring $R/(qR)$. Whenever working in the quotient ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ and $R_{2q} = Z_{2q}[x]/(x^n + 1)$, we will assume that n is a power of 2 and q is a prime number such that $q = 1 \mod 2n$. We define the rotation matrix of a polynomial $\mathbf{a} \in R_q$ as $rot(\mathbf{a}) = (\mathbf{a}, \mathbf{a} \cdot \mathbf{x}, \mathbf{a} \cdot \mathbf{x}^2, \ldots, \mathbf{a} \cdot \mathbf{x}^{n-1}) \in \mathbb{Z}_q^{n \times n}$. Then for $\mathbf{a}, \mathbf{b} \in R_q$, the matrix-vector product $rot(\mathbf{a}) \cdot \mathbf{b} \mod q$ corresponds to the product of polynomials $\mathbf{ab} \in R_q$. We write $\| \cdot \|$ for the Euclidean norm.

5.2.2 Lattices and Bases

In this chapter, we use the following definition of lattices. Let m be a positive integer and let $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\} \subset \mathbb{R}^m$ be a set of linearly independent vectors. The lattice $\mathtt{L}$ generated by $\mathbf{B}$ is the set of linear combinations of $\mathbf{b}_1, \ldots, \mathbf{b}_n$ with coefficients in $\mathbb{Z}$, viz.

$$\mathtt{L}(\mathbf{B}) = \left\{ \mathbf{x} \in \mathbb{R}^m | \ \mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{b_i} \text{ for some } \alpha_i \in \mathbb{Z} \right\}.$$

A basis for $\mathtt{L}$ is any set of linearly independent vectors that generates $\mathtt{L}$. Any two such sets have the same number of elements. The dimension of $\mathtt{L}$ is the number of vectors in a basis for $\mathtt{L}$. The rank of the lattice $\mathtt{L}$ is defined to be the rank of the matrix B. If the rank equals m we say that $\mathtt{L}$ is full-rank. Abusing notation, we identify lattice bases with matrices and vice versa by taking the basis vectors as the columns of the matrix. The length of the shortest non-zero vectors of a lattice $\mathtt{L}$ is denoted by $\lambda_1(\mathtt{L})$. Let q be a positive integer. A lattice $\mathtt{L}$ that contains $q\mathbb{Z}^m$ is called a q-ary lattice. For a matrix

$\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, we define the q-ary lattice.

$$\mathtt{L}_q(\mathbf{A}) = \{\mathbf{v} \in \mathbb{Z}^m | \ \exists \mathbf{w} \in \mathbb{Z}^n : \mathbf{A}\mathbf{w} = \mathbf{v} \mod q\}.$$

For a lattice basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\} \subset \mathbb{R}^m$ its fundamental parallelepiped is defined as

$$P(\mathbf{B}) = \left\{ \mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{b}_i \in \mathbb{R}^m \mid \forall i : 0 \leq \alpha_i < 1 \right\}.$$

The determinant $det(\mathtt{L})$ of a lattice $\mathtt{L} \subset \mathbb{R}^m$ is defined as the m-dimensional volume of the fundamental parallelepiped of a basis of $\mathtt{L}$. Note that the determinant of the lattice is well-defined, i.e. it is independent of the basis. The Hermite factor δ of a lattice basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\} \subset \mathbb{R}^m$ is defined via the equation $\|b_1\| = \delta^m det(\mathtt{L})^{1/m}$. It provides a measure for the quality of the basis (for more details, we refer to [33]).

Let $\mathtt{L}$ be a lattice and let $\mathbf{x}$ be a point. Let $\mathbf{y} \in \mathtt{L}$ be the lattice point for which the length $\|\mathbf{x} - \mathbf{y}\|$ is minimised. We define the distance to the lattice $\mathtt{L}$ from the point $\mathbf{x}$ to be this length, which we denote $dist(\mathbf{x}, \mathtt{L})$.

Lattice-based cryptography is based on the presumed hardness of computational problems in lattices. Two of the most important lattice problems are defined next.

5.2.2.1 Shortest Vector Problem (SVP)

The task is to find a shortest nonzero vector in a lattice $\mathtt{L}$, i.e. find a non-zero vector $\mathbf{v} \in \mathtt{L}(\mathbf{B})$ that minimises the Euclidean norm $\|\mathbf{v}\|$, i.e. such that $\|\mathbf{v}\| = \lambda_1(\mathtt{L}(\mathbf{B}))$.

5.2.2.2 Bounded Distance Decoding (BDD)

Given $\alpha \in \mathbb{R}_{\geq 0}$, a lattice basis $\mathbf{B}$, and a target vector $\mathbf{t} \in \mathbb{R}^m$ with $dist(\mathbf{t}, \mathtt{L}(\mathbf{B})) < \alpha \cdot \lambda_1(\mathtt{L}(\mathbf{B}))$, the goal is to find a vector $\mathbf{e} \in \mathbb{R}^m$ with $\|\mathbf{e}\| < \alpha\lambda_1(\mathtt{L}(\mathbf{B}))$ such that $\mathbf{t} - \mathbf{e} \in \mathtt{L}(\mathbf{B})$.

5.2.2.3 Babai's Nearest Plane Algorithm

This algorithm was introduced in [5] and it has been utilised as an oracle to carry out various attacks [35, 16, 72]. The input for Babai's Nearest Plane algorithm is a lattice basis $\mathbf{B} \in \mathbb{Z}^m$ and a target vector $\mathbf{t} \in \mathbb{R}^m$ and the corresponding output is a vector $\mathbf{e} \in \mathbb{R}^m$ such that $\mathbf{t} - \mathbf{e} \in \mathtt{L}(\mathbf{B})$. We denote the output by $\mathbf{NP}_{\mathbf{B}}(\mathbf{t}) = \mathbf{e}$.

If the lattice basis $\mathbf{B}$ consists of vectors that are pairwise orthogonal, i.e., such that $\mathbf{v}_i \cdot \mathbf{v}_j = 0$ for all $i \neq j$, then it is very easy to solve both SVP and BDD. In general, if the vectors in the basis are reasonably orthogonal to one another, then Babai's Nearest Plane Algorithm may solve BDD, but if the basis vectors are highly nonorthogonal, then the vector returned by the algorithm is generally far from the closest lattice vector to $\mathbf{t}$. In such a case, before applying the algorithm, we can reduce the lattice basis $\mathbf{B}$ to get a "better" basis by using some basis reduction algorithm [33]. For example, Lenstra-Lenstra-Lovász (LLL) algorithm or the block Korkin-Zolotarev variant of the LLL algorithm (BKZ-LLL). These algorithms produce a basis in which the basis vectors are quasi-orthogonal, i.e., they are reasonably orthogonal to one another.

5.3 BLISS Signature Scheme

In this section we briefly describe the BLISS key generation algorithm [22].

5.3.1 The BLISS Key Generation Algorithm

Given two real numbers $0 \leq \delta_1 < 1$ and $0 \leq \delta_2 < 1$, two random polynomials $\mathbf{f}$ and $\mathbf{g}$ are generated such that both polynomials have $d_1 = \lceil \delta_1 n \rceil$ coefficients in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ coefficients in $\{\pm 2\}$, assuring that $\mathbf{f}$ is invertible. The secret key is given by $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t = (\mathbf{f}, 2\mathbf{g} + 1)^t$.

The public key is then computed as follows: set $\mathbf{a_q} = (2\mathbf{g} + 1)/\mathbf{f} \in R_q$. Next, define $\mathbf{A} = (2 \cdot \mathbf{a_q}, q - 2) \in R_{2q}^{1\times 2}$. One easily verifies that:

$$\mathbf{AS} = 2\mathbf{a_q} \cdot \mathbf{f} - 2(2\mathbf{g} + 1) = 0 \qquad \mod q,$$

$$\mathbf{AS} = q(2\mathbf{g} + 1) = q \cdot 1 = 1 \qquad \mod 2.$$

That is $\mathbf{AS} = q \mod 2q$. Finally, $(\mathbf{A}, \mathbf{S})$ is a valid key pair for the scheme.

5.4 The strongSwan Project

The strongSwan project is a multi-platform, open-source IPSec implementation. As its website describes,[1] this project was originally based on the discontinued FreeS/WAN project and continues to be released under the GPL license. This project was launched in 2005 to provide a stable and open source IPsec implementation, and was originally designed for Linux, although it has since been ported to other platforms, such as Android, FreeBSD, Mac OS X and Windows. It offers simplicity of configuration, strong cryptographic algorithms, set-up of IPsec policies to give support to large and complex VPN networks, and other features.

5.4.1 The strongSwan BLISS Implementation

Starting with the strongSwan 5.2.2 release, BLISS is offered as an IKEv2 public key authentication method. Full BLISS key and certificate generation support is also added to the strongSwan PKI tool. With strongSwan 5.3.0, there was an upgrade to the improved BLISS-B signature algorithm described in [23]. A recent research paper by Pessl *et al.* [58] explores cache attacks on this particular implementation and shows that cache attacks on this signature scheme are not only possible, but also practical. Its authors claim their attack recovers the secret signing key after observing roughly 6000 signature generations. However, there is no research paper, to the best of our knowledge, exploring cold boot attacks on this implementation.

We will next describe the private key in-memory format used by this implementation.

Private Key In-Memory Format. Figure 5.1 shows the definition of the C struct private_bliss_private_key_t extracted from the file bliss_private_key.c. It defines

[1] See https://www.strongswan.org/ for details of this project.

```
/**
 * BLISS secret key S1 (coefficients of polynomial f)
 */
int8_t *s1;

/**
 * BLISS secret key S2 (coefficients of polynomial 2g + 1)
 */
int8_t *s2;

/**
 * NTT of BLISS public key (coefficients of polynomial (2g + 1)/f)
 */
uint32_t *A;

/**
 * NTT of BLISS public key in Montgomery representation Ar = rA mod
 */
uint32_t *Ar;
```

Figure 5.1: struct `private_bliss_private_key_t`

the data structure used to store the private key after the subroutine `bliss_private_key_gen` defined in the file `bliss_private_key.c` has been executed. When this particular subroutine is executing, it invokes internally the subroutine `create_vector_from_seed` to create both $\mathbf{f}$ and $\mathbf{g}$.

Each of these polynomials has d_1 coefficients with values in the set $\{-1, 1\}$, d_2 coefficients with values belonging the set $\{-2, 2\}$ and its remaining coefficients have values equal to zero. Additionally, $\mathbf{f}$ is chosen to be invertible in R_q. The algorithm also computes both the polynomial $2\mathbf{g} + \mathbf{1}$ and the public polynomial $\mathbf{a_q} = (2\mathbf{g} + \mathbf{1})/\mathbf{f}$. The variable `s1` will point to an array whose entries are the coefficient of the polynomial $\mathbf{f}$ while the variable `s2` will point to an array whose entries are the coefficients of polynomial $2\mathbf{g} + \mathbf{1}$. Each entry of either of the two arrays is an 8-bit integer.

Parameter sets. This implementation includes 4 named reference parameter sets with a range of choices for n, q, d_1 and d_2, targeting different security levels and optimisations. These parameter sets are defined in the file `bliss_param_set.c` and were proposed by the BLISS designers in [22]. For example, the parameter set `BLISS-I` targets 128 bits of security and defines $n = 512$, $q = 12289$, $d_1 = 154$, $d_2 = 0$, whilst the parameter set `BLISS-IV` targets 192 bits of security and defines $n = 512$, $q = 12289$, $d_1 = 231$, $d_2 = 31$.

5.5 Mounting Cold Boot Key Recovery Attacks

In this section, we present several cold boot key recovery attacks on the `strongSwan` BLISS implementation and its corresponding private key format which was introduced in the previous section.

5.5.1 Initial Observations

We continue to make the assumptions outlined in Section 2.3. Our cold boot attack model assumes that the adversary can obtain a noisy version of the original BLISS private key (using whatever format is used to store it in memory). We assume that the corresponding BLISS public key is known exactly (without noise). We additionally assume that all relevant public parameters and private key formatting information are known to the adversary. Our aim is then recover the private key.

In particular, we assume the attacker obtains $\mathtt{s}'_1$ and $\mathtt{s}'_{\mathbf{2}}$, which are the noisy versions of the arrays storing $\mathbf{f}$ and $2\mathbf{g} + \mathbf{1}$ respectively. Both the array $\mathtt{s}'_1$ and the array $\mathtt{s}'_2$ have n entries. We also assume the attacker knows that $\mathbf{a_q} \cdot \mathbf{f} = 2\mathbf{g} + \mathbf{1}$ and the parameters that were used to create the private polynomials, i.e. the values of d_1 and d_2. This is a plausible assumption because the values d_1 and d_2 can be found in a public file.

According to the key generation algorithm, both the polynomial $\mathbf{f}$ and the polynomial $\mathbf{g}$ have d_1 coefficients with values in the set $\{-1, 1\}$, d_2 coefficients with values in the set $\{-2, 2\}$ and their remaining coefficients have value zero. Therefore, the polynomial $\mathbf{h} = 2\mathbf{g} + \mathbf{1}$ has its constant coefficient value h_0 in the set $\{-1, -3, 1, 3, 5\}$ and satisfies:

1. If h_0 is 1, then $\mathbf{h}$ has d_1 coefficients with values in the set $\{-2, 2\}$, d_2 coefficients with values in the set $\{-4, 4\}$ and its remaining coefficients have value zero.
2. If $h_0 \in \{-1, 3\}$, then $\mathbf{h}$ has $d_1 - 1$ coefficients with values in the set $\{-2, 2\}$, d_2 coefficients with values in the set $\{-4, 4\}$ and its remaining coefficients have value zero.
3. If $h_0 \in \{-3, 5\}$, then $\mathbf{h}$ has d_1 coefficients with values in the set $\{-2, 2\}$ and $d_2 - 1$ coefficients with values in the set $\{-4, 4\}$ and its remaining coefficients have value zero.

We will make use of these properties of the polynomials $\mathbf{f}$ and $\mathbf{h}$ to design key recovery algorithms.

5.5.2 Key Recovery Via Key Enumeration

In this section we will develop algorithms for tackling the problem of recovering $\mathbf{f}$ and $\mathbf{h}$ from $\mathbf{s}'_1$ and $\mathbf{s}'_\mathbf{2}$ by running instances of key enumeration algorithms, thoroughly discussed in Chapter 3, as well as exploiting the underlying properties of the polynomials $\mathbf{f}$ and $\mathbf{h}$.

The goal of the attacker is to search for key candidates of the form $(\mathbf{f}_i, \mathbf{h}_j)$ for the 2-tuple $(\mathbf{f}, \mathbf{h})$. To do so, the attacker will first need to construct both a list L_f that contains high scoring key candidates for the polynomial $\mathbf{f}$ and a list L_h that contains high scoring key candidates for the polynomial $\mathbf{h}$. Next the attacker will take both a key candidate from the first list and a key candidate from the second list and finally verify if such pair may be the real private key.

The first task of the attacker will then be producing both the list L_f and the list L_h. Because both the list L_f and the list L_h must store high scoring key candidates for the polynomials $\mathbf{f}$ and $\mathbf{h}$ respectively, the attacker may make use of the key recovery strategy introduced in Section 4.4.1 to create such lists as follows.

5.5.2.1 Constructing L_f from $\mathbf{s}'_1$

Recall that the attacker has access to $\mathbf{s}'_1$ that is a bit string of size $W = 8n$. Let us set $\mathbf{r} = \mathbf{s}'_1$. The attacker first sets a chunk to be an 8-bit string, i.e. $w = 8$, and so there are $n = W/w$ chunks. For a given chunk, its candidates are then the integers in the set $\{-2, -1, 0, 1, 2\}$. The attacker now sets a block to be a successive sequence of $l > 0$ chunks, with $l \mid n$, and so there are $n_b = n/l$ blocks. To construct the list L_f, the attacker then proceeds as follows.

`Phase I`

1. For each chunk $\mathbf{r}^i$, $0 \leq i < n$, the attacker uses Equation (2.1) to compute log-

likelihood scores for each candidate $\mathtt{c}^i$ for the chunk, viz.

$$\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i] := n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta),$$

where the n^i_{ab} values count occurrences of bits across the i-th chunks, $\mathtt{r}^i$, $\mathtt{c}^i$. So the attacker obtains a list of chunk candidates with 5 entries for the chunk, since the candidates for any chunk are the integers in the set $\{-2, -1, 0, 1, 2\}$.

2. For each block $\mathtt{b}^j$, $0 \leq j < n_b$, the attacker presents the l lists corresponding to the l chunks in the block $\mathtt{b}^j$ as inputs to OKEA to produce a list with the M_j highest scoring chunk candidates for the block, L_{b^j}.

Once `Phase I` is completed, the attacker will obtain n_b lists of chunk candidates and proceed as follows.

`Phase II`

1. The attacker presents the n_b lists as inputs to a key enumeration algorithm in order to generate M_f high scoring key candidates $\mathbf{f}_i$ for $\mathbf{f}$, which are then stored in a list L_f. The instance of the key enumeration algorithm only outputs valid candidate polynomials $\mathbf{f}_i$ for $\mathbf{f}$, i.e. those having d_1 coefficients with values in the set $\{1, -1\}$, d_2 coefficients with values in the set $\{2, -2\}$ and the remaining coefficients with value zero. With respect to the verification of $\mathbf{f}_i$, it can be simply done by running through each entry of the array representing $\mathbf{f}_i$, while counting the array's entries with values in the set $\{-1, 1\}$, its entries with values in the set $\{-2, 2\}$ and its entries with value zero. If the three counting variables are equal to d_1, d_2 and $n - d_1 - d_2$ respectively, then $\mathbf{f}_i$ is a valid candidate for $\mathbf{f}$. It is preferable to run this check at this point because during and immediately after `Phase I` the algorithm can not control or know what entries will have values either in the set $\{-1, 1\}$ or in the set $\{-2, 2\}$ or value zero.

5.5.2.2 Constructing L_h from $\mathtt{s}'_2$

Similarly the attacker has access to $\mathtt{s}'_2$, a bit string with $W = 8 \cdot n$ bits. Let us set $\mathtt{r} = \mathtt{s}'_2$. The attacker then sets a chunk to be a bit-string of size $w = 8$, so there are $W/w = n$ chunks. Note that the candidates for the first chunk are the integers in the

set $\{-3,-1,1,3,5\}$, while the candidates for the remaining chunks are the integers in the set $\{-4,-2,0,2,4\}$. The attacker now sets a block to be a consecutive sequence of $l > 0$ chunks, with $l \mid n$, and so there are $n_b = n/l$ blocks. To construct the list L_h, the attacker then proceeds as follows.

`Phase I`

1. For each chunk $\mathtt{r}^i$, the attacker uses Equation (2.1) to compute log-likelihood scores $\mathcal{L}[\mathtt{c}^i;\mathtt{r}^i]$ for each candidate $\mathtt{c}^i$ for the chunk. So the attacker obtains a list of chunk candidates with 5 entries for each chunk, because the candidates for the first chunk are the integers in the set $\{-3,-1,1,3,5\}$, while the candidates for the remaining chunks are the integers in the set $\{-4,-2,0,2,4\}$.
2. For each block $\mathtt{b}^j$, $0 \leq j < n_b$, the attacker presents the l lists corresponding to the l chunks in the block $\mathtt{b}^j$ as inputs to OKEA to produce a list with the M_j highest scoring chunk candidates for the block, L_{b^j}.

Once `Phase I` is completed, the attacker will obtain n_b lists of chunk candidates and proceed as follows.

`Phase II`

1. The attacker presents the n_b lists as inputs to a key enumeration algorithm in order to generate M_h high scoring key candidates $\mathbf{h}_i$ for $2\mathbf{g}+\mathbf{1}$, which are then stored in a list L_h. This instance of the key enumeration algorithm only outputs valid candidate polynomials $\mathbf{h}_i$ for $2\mathbf{g}+\mathbf{1}$, i.e. those having the properties described in Section 5.5.1. The verification of $\mathbf{h}_i$, it can be simply done by computing $\mathbf{h}'_i = \mathbf{h}_i - \mathbf{1}$ and then running through each entry of the array representing $\mathbf{h}'_i$ while counting its entries with values in the set $\{-2,2\}$, its entries with values in the set $\{-4,4\}$ and its entries with values zero. If the three counting variables are equal to d_1, d_2 and $n-d_1-d_2$ respectively, then $\mathbf{h}_i$ is a valid candidate for $2\mathbf{g}+\mathbf{1}$. Similarly, it is preferable to run this check at this point for similar reasons to those posed for the other case.

5.5.2.3 Combining L_f and L_h

Once both the list L_f and the list L_h are created, the attacker will run another instance of a key enumeration algorithm receiving both the list L_f and the list L_h as input, which then proceeds to find high scoring candidates for $(\mathbf{f}, 2\mathbf{g} + \mathbf{1})$ that are obtained by taking an $\mathbf{f}_i$ from L_f and an $\mathbf{h}_j$ from L_h, whilst ensuring that $\mathbf{a_q} \cdot \mathbf{f}_i = \mathbf{h}_j$, i.e. that there is a collision of polynomials $\mathbf{a_q} \cdot \mathbf{f}_i$ and $\mathbf{h}_j$ in the two lists. If such a collision is found, then $(\mathbf{f}_i, \mathbf{h}_j)$ is a valid candidate pair for $(\mathbf{f}, 2\mathbf{g} + \mathbf{1})$; otherwise it is not.

Regarding the overall performance of the previous key recovery strategy, we note that the processes for creating the lists L_f and L_h can be run simultaneously because the dependence of the two on each other is loose. In the combining phase, it is more convenient to run a non-optimal enumeration algorithm (see Chapter 3 for more details), because the search can be parallelized and hence performed more efficiently.

With regard to how successful the previous key recovery strategy might be, we note that the key recovery strategy will find a valid pair $(\mathbf{f}_i, \mathbf{h}_j)$ if and only if there is an $\mathbf{f}_i \in L_f$ and an $\mathbf{h}_j \in L_h$ with the property that $\mathbf{a_q} \cdot \mathbf{f}_i = \mathbf{h}_j$. Hence its success probability will improve by increasing the sizes of L_f and L_h. This can be problematic in terms of memory and performance when M_f and M_h grows bigger, leading us to make refinements to it as follows.

We will make use of a hash table $\mathcal{T}_1$. A hash table is a data structure used to map keys to values. It uses a hash function to calculate an index in an array of slots, from which the desired value may be found. Ideally, the hash function should assign each key to a unique slot, but most hash table designs employ an imperfect hash function, which might cause hash collisions, i.e., the hash function generates the same index for more than one key. Hence, we have to use effective techniques for resolving the conflict created by collisions. The simplest collision resolution technique is chaining. In chaining, we place all the elements whose keys hash to the same slot into the same linked list pointed to by the slot. This is, a slot of index t contains a pointer to the head of the linked list of all stored elements whose keys hash to t [17].

The attacker now generates M_h high scoring key candidates $\mathbf{h}_i$ for $2\mathbf{g} + \mathbf{1}$ by following a similar process as described in Section 5.5.2.2, but instead of storing them in a list L_h, the

attacker will store them in the global hash table $\mathcal{T}_1$. For each $\mathbf{h}_i$, the attacker computes a hash value k_{h_i} (used as a key) from all the entries of the array representation $\mathtt{h}_i$ of $\mathbf{h}_i$, then calculates an index t_{h_i} from k_{h_i} by using a map I and finally adds $\mathtt{h}_i$ as the first entry of a linked list pointed to by the entry $\mathcal{T}_1[t_{h_i}]$. Note that the number of operations for adding an array to the table $\mathcal{T}_1$ is bounded by a constant C_1 on average. Therefore, generating the table $\mathcal{T}_1$ costs $M_h \cdot C_1$ operations on average.

Once this is done, the attacker will generate valid and high scoring key candidates $\mathbf{f}_i$ for $\mathbf{f}$ as described previously. For each $\mathbf{f}_i$, the algorithm calculates a hash value k_{af_i} (used as a key) from all the entries of the array representation $\mathtt{af}_i$ of $\mathbf{a_q} \cdot \mathbf{f}_i$, then computes the index $t_{af_i} = I(k_{af_i})$ and finally checks if $\mathtt{af}_i$ can be found in the linked list pointed to by the entry $\mathcal{T}_1[t_{af_i}]$ (if exists). If a match is found, then $\mathbf{a_q} \cdot \mathbf{f}_i = \mathbf{h}_j$, producing a valid candidate pair for $(\mathbf{f}, 2\mathbf{g} + \mathbf{1})$. Note that it is expected that the number of entries of any linked list pointed to by any entry of $\mathcal{T}_1$ is bounded by a constant C_2, assuming the function I disperses the elements properly among the buckets, so searching through $\mathcal{T}_1[t_{af_i}]$ is also expected to be done in a constant number of operations on average.

By using just a hash table, this variant scales better in terms of memory than the previous key recovery strategy. As far as performance is concerned, there is also less restriction on generating high scoring candidates for $2\mathbf{g}+\mathbf{1}$, because the order in which they are generated during an enumeration is irrelevant; what is important is that they are included in the global hash table. Hence, the attacker may run an instance of a non-optimal enumeration algorithm to generate candidates for $2\mathbf{g} + \mathbf{1}$, e.g. by selecting a suitable interval in which either the candidates' scores lie or the candidates' ranks lie (see algorithms described in Section 3.2.3 and Section 3.2.5 respectively). With regard to the generation of candidates for $\mathbf{f}$, it may also be carried out by running a non-optimal enumeration algorithm since it can then be customised to perform a more efficient search (see Chapter 3 for more details).

5.5.2.4 Enumerating Only Candidates for f

By exploiting further the relation $\mathbf{a_q} \cdot \mathbf{f} = 2\mathbf{g} + \mathbf{1}$ and the properties of $2\mathbf{g} + \mathbf{1}$ as a filter, the attacker can just enumerate candidates for $\mathbf{f}$ and not store any tables in memory. We explain how this can be done next.

By following a similar procedure as described in Section 5.5.2.1, the attacker can generate

key candidates $\mathbf{f}_i$ for $\mathbf{f}$ from $\mathbf{s}'_1$. For each output $\mathbf{f}_i$, the attacker computes the polynomial $\mathbf{h}_i = \mathbf{a_q} \cdot \mathbf{f}_i$ and

1. Verifies if its constant coefficient value is 1 and it has d_1 coefficient values in $\{-2, 2\}$ and d_2 coefficient values in $\{-4, 4\}$ and its remaining coefficients values are zero. If so, then the pair $(\mathbf{f}_i, \mathbf{h}_i)$ has been found. Otherwise, go to 2.
2. Verifies if its constant coefficient value is in $\{-1, 3\}$ and it has $d_1 - 1$ coefficient values in $\{-2, 2\}$ and d_2 coefficient values in $\{-4, 4\}$ and its remaining coefficients are zero. If so, then the pair $(\mathbf{f}_i, \mathbf{h}_i)$ has been found. Otherwise, go to 3.
3. Verifies if its constant coefficient value is in $\{-3, 5\}$ and it has d_1 coefficient values in $\{-2, 2\}$ and $d_2 - 1$ coefficient values in $\{-4, 4\}$ and its remaining coefficients are zero. If so, then the pair $(\mathbf{f}_i, \mathbf{h}_i)$ has been found, or else keep searching for a suitable $\mathbf{f}_i$.

If the algorithm finds a pair $(\mathbf{f}_i, \mathbf{h}_i)$, then it is a candidate pair for $(\mathbf{f}, 2\mathbf{g} + \mathbf{1})$. In fact, it is very likely to be the correct private key. To see why, let $\mathcal{P}$ be the set of all polynomials having d_1 coefficients values in $\{-1, 1\}$, d_2 coefficient values in $\{-2, 2\}$ and the remaining coefficients values zero. We ask for the probability that some random $\mathbf{f} \in \mathcal{P}$ has the property that $\mathbf{h} = \mathbf{f} \cdot \mathbf{a_q} - \mathbf{1} \in R_q$ is a polynomial having d_1 coefficients with values in the set $\{-2, 2\}$, d_2 coefficients with values in the set $\{-4, 4\}$ and its remaining coefficients have value zero. Treating the coefficients of $\mathbf{h}$ as independent random variables that are uniformly distributed modulo q, the probability that any particular coefficient is in the set either $\{-2, 2\}$ or $\{-4, 4\}$ is $2/q$, and that any particular coefficient is zero is $1/q$, then the probability we are asking for is about $\frac{n!}{d_1!d_2!(n-d_1-d_2)!}(2/q)^{d_1+d_2}(1/q)^{n-d_1-d_2}$, which is negligible for common parameters. Therefore, this makes it unlikely that the approach will find invalid candidates. For example, let us take the parameter set `BLISS-I` with $n = 512$, $q = 12289$, $d_1 = 154$, $d_2 = 0$. In such case the probability we are asking for is about

$$\frac{512!}{154!0!(358)!}(2/12289)^{154}(1/12289)^{358} \approx 1.226 \times 10^{-1913}.$$

If we now take the parameter set `BLISS-IV` with $n = 512$, $q = 12289$, $d_1 = 231$, $d_2 = 31$, then the probability is about

$$\frac{512!}{231!31!(250)!}(2/12289)^{262}(1/12289)^{250} \approx 7.954 \times 10^{-1823}.$$

On the other hand, as far as performance is concerned, the attacker needs to choose the

key enumeration algorithm to run in `Phase II` for generating candidates $\mathbf{f}_i$ very carefully. In this setting, OKEA would not be suitable, since it is inherently a serial algorithm and requires a lot of memory (see Section 3.2.1.4). Instead, non-optimal key enumeration algorithms would suit better in this setting, since they are parallelizable and memory-efficient. Even though their outputs may not be given in the optimal order, the search can be customisable by either selecting a suitable interval in which the outputs' scores lie, e.g. the key enumeration algorithm described in Section 3.2.3 or selecting a suitable interval in which the outputs' ranks lie, e.g. the key enumeration algorithm described in Section 3.2.5. Therefore, we will use a non-optimal key enumeration algorithm during `Phase II`. This algorithm will be a novel non-optimal key enumeration algorithm that combines key features from various non-optimal key enumeration algorithms introduced in Chapter 3 and will be described in Section 5.6.2.3.

Regarding the verification of $\mathbf{h}_i$, it can be simply done by computing $\mathbf{h}'_i = \mathbf{h}_i - \mathbf{1}$ and then running through each entry of the array representing $\mathbf{h}'_i$ while counting its entries with values in the set $\{-2, 2\}$, its entries with values in the set $\{-4, 4\}$ and its entries with values zero. If the three counting variables are equal to d_1, d_2 and $n - d_1 - d_2$ respectively, then $\mathbf{h}_i$ is a valid candidate for $2\mathbf{g} + \mathbf{1}$. We will experimentally evaluate this key recovery algorithm in Section 5.6.2.

5.5.3 Casting the Problem as an LWE Instance

In this section, we establish a connection between the key recovery problem in this particular case with a non-conventional instance of Learning with Errors Problem (LWE).

Let us consider the polynomials $\mathbf{s}'_1, \mathbf{s}'_2 \in R_q$ obtained from the noisy arrays $\mathbf{s}'_1$ and $\mathbf{s}'_2$ respectively. We can re-write $\mathbf{s}'_1$ as $\mathbf{s}'_1 = \mathbf{f} + \mathbf{e}_1$ and $\mathbf{s}'_2$ as $\mathbf{s}'_2 = 2\mathbf{g} + \mathbf{1} + \mathbf{e}_2$, where $\mathbf{e}_1, \mathbf{e}_2$ are error polynomials. Hence, we have

$$\mathbf{a_q} \cdot \mathbf{s}'_1 = \mathbf{a_q} \cdot (\mathbf{f} + \mathbf{e}_1) = \mathbf{a_q} \cdot \mathbf{f} + \mathbf{a_q} \cdot \mathbf{e}_1 = 2\mathbf{g} + 1 + \mathbf{a_q} \cdot \mathbf{e_1} = \mathbf{s}'_2 - \mathbf{e}_2 + \mathbf{a_q} \cdot \mathbf{e}_1.$$

Therefore

$$\mathbf{a_q} \cdot \mathbf{s}'_1 - \mathbf{s}'_2 = \mathbf{a_q} \cdot \mathbf{e}_1 - \mathbf{e}_2 \in R_q. \tag{5.1}$$

It is clear that the left-hand side of Equation (5.1) can be computed by the attacker, but the attacker does not learn any additional information on the nature of the values the coefficients of $\mathbf{e}_1$ and $\mathbf{e}_2$ may have. Now if the attacker computes a high scoring candidate $\mathbf{c}_1$ from $\mathbf{s}'_1$ for $\mathbf{f}$ by using algorithm described in Section 5.5.2.1 and a high candidate $\mathbf{c}_2$ from $\mathbf{s}'_2$ for $2\mathbf{g}+\mathbf{1}$ by using algorithm described in Section 5.5.2.2, then there will be also polynomials $\mathbf{e}'_1$ and $\mathbf{e}'_2$ such that $\mathbf{c}_1 = \mathbf{f} + \mathbf{e}'_1$ and $\mathbf{c}_2 = 2\mathbf{g} + \mathbf{1} + \mathbf{e}'_2$. Therefore, the above equation can be rewritten as

$$\mathbf{a_q} \cdot \mathbf{c}_1 - \mathbf{c}_2 = \mathbf{a_q} \cdot \mathbf{e}'_1 - \mathbf{e}'_2 \in R_q, \tag{5.2}$$

where the polynomials $\mathbf{e}'_1$, $\mathbf{e}'_2$ probably have many coefficients with value zero and their respective non-zero coefficient values are small. Indeed, since $\mathbf{c}_1$ was chosen to be a valid candidate polynomial for $\mathbf{f}$, it follows that the coefficients of $\mathbf{e}'_1$ have values in the set $V_1 = \{-4, -3, \ldots, 3, 4\}$. Similarly, since $\mathbf{c}_2$ was chosen to be a valid candidate polynomial for $2\mathbf{g}+\mathbf{1}$, then the coefficients of $\mathbf{e}'_2$ have values in the set $V_2 = \{-8, -6, \ldots, 6, 8\}$. We can now re-write Equation (5.2) as

$$\mathbf{c} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \mod q, \tag{5.3}$$

where $\mathbf{c} \in \mathbb{Z}_q^n$ is the column vector associated with the polynomial $\mathbf{a_q} \cdot \mathbf{c}_1 - \mathbf{c}_2$, $\mathbf{s} \in \mathbb{Z}_q^n$ is the column vector associated with the polynomial $\mathbf{e}'_1$, $\mathbf{e} \in \mathbb{Z}_q^n$ is the column vector associated with the polynomial $-\mathbf{e}'_2$ and $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ is the rotation matrix of $\mathbf{a_q}$, i.e.

$$\mathbf{A} = \begin{pmatrix} a_0 & -a_{n-1} & \cdots & -a_1 \\ a_1 & a_0 & \cdots & -a_2 \\ \vdots & \vdots & \cdots & \vdots \\ a_{n-1} & a_{n-2} & \cdots & a_0 \end{pmatrix}.$$

This shows that given a high-scoring candidate $(\mathbf{c}_1, \mathbf{c}_2)$, the problem of recovering $\mathbf{f}$ and $\mathbf{h}$ can be re-cast as a Ring-LWE instance (since if $\mathbf{s}$ can be found, then so can $\mathbf{f}$ and $\mathbf{h}$).

However, the noise distribution in this instance, arising from $\mathbf{e}$, is not the usual one in the LWE setting [59].

5.5.3.1 Meet-in-the-Middle Attacks

Here we will explore standard meet-in-the-middle attacks to solve LWE. This is a time-memory trade-off approach and is therefore a faster method than a naive brute force at the cost of an increased memory requirement [59].

The attacker starts off by splitting $\mathbf{s} = \left(\mathbf{s}^l|\mathbf{s}^r\right)$ and $\mathbf{A} = \left(\mathbf{A}^l|\mathbf{A}^r\right)$ into two parts and rewriting Equation (5.3) as

$$\mathbf{c} = \mathbf{A}^l \cdot \mathbf{s}^l + \mathbf{A}^r \cdot \mathbf{s}^r + \mathbf{e} \mod q, \tag{5.4}$$

where $\mathbf{s}^l \in V_1^k \subset \mathbb{Z}_q^k$, $\mathbf{s}^r \in V_1^{n-k} \subset \mathbb{Z}_q^{n-k}$, $\mathbf{A}^l \in \mathbb{Z}_q^{n\times k}$, $\mathbf{A}^r \in \mathbb{Z}_q^{n\times(n-k)}$ and $\mathbf{e} \in V_2^n \subset \mathbb{Z}_q^n$.

Note that if the attacker guessed the correct vector $\mathbf{s}^l$ and the correct vector $\mathbf{s}^r$, then the vector $\mathbf{A}^l \cdot \mathbf{s}^l$ would be "almost" equal to $\mathbf{c} - \mathbf{A}^r \cdot \mathbf{s}^r$, because calculating $\mathbf{A}^l \cdot \mathbf{s}^l + \mathbf{e}$ would likely not make $\mathbf{A}^l \cdot \mathbf{s}^l$ to change considerably, since $\mathbf{e}$ is very likely to have many zero entries and its non-zeros entries are in V_2. In other words, $\mathbf{c} - \mathbf{A}^r \cdot \mathbf{s}^r$ and $\mathbf{A}^l \cdot \mathbf{s}^l$ differ only by a vector $\mathbf{e}$ such that $\max(|e_1|, |e_2|, \ldots, |e_n|) \leq 8$.

The attacker therefore may be able to choose some b such that given $\mathbf{u} = \mathbf{A}^l \cdot \mathbf{s}_i^l$ and $\mathbf{v} = \mathbf{c} - \mathbf{A}^r \cdot \mathbf{s}_j^r$, then $MSB_b(u_p) = MSB_b(v_p)$ with overwhelming probability for all $0 \leq p \leq n_1 \leq n-1$, where $MSB_b(x)$ denotes the b most significant bits of the binary representation of x. Let us define a hash function H such that $H(\mathbf{z})$ outputs a hash value involving all the n_1 $MSB_b(z_i)$ values. Therefore, given $\mathbf{u}$ and $\mathbf{v}$ satisfying that $MSB_b(u_p) = MSB_b(v_p)$ for all $0 \leq p \leq n_1$, then $H(\mathbf{u}) = H(\mathbf{v})$. Note the attacker may use H to search for collisions. Indeed, the attacker may proceed to recover $\mathbf{s}$ by using H and hash table $\mathcal{T}_2$, similar to the hash table $\mathcal{T}_1$ described previously, as follows.

The attacker first computes guesses for $\mathbf{s}^l$. For each candidate $\mathbf{s}_i^l \in V_1^k$, the attacker first computes $\mathbf{u}_i^l = \mathbf{A}^l \cdot \mathbf{s}_i^l$ and then adds the array representation of $\mathbf{s}_i^l$ as the first entry of a linked list pointed to by $\mathcal{T}_2[t_i^l]$, calculating t_i^l from $H(\mathbf{u}_i^l)$ via using a map I outputting indexes in the hash table $\mathcal{T}_2$ from hash values.

To enumerate all possible candidates in V_1^k, the attacker first generates all the j-combinations of k for $0 \leq j \leq k$, where a j-combination of k is defined as a subset of j distinct elements of the set $\{0, 2, 3, \ldots, k-1\}$ and is represented as $(c_0, c_1, \ldots c_{j-1})$. For each j-combination $(c_0, c_1, \ldots c_{j-1})$, the attacker generates $(|V_1| - 1)^j$ vectors of size k by running a modified version of Algorithm 16 from Section 3.2.5. This is, the tweaked algorithm picks j values out of $V_1 \setminus \{0\}$, assign them to the corresponding entries indexed by $c_0, c_1, \ldots, c_{j-1}$ and set the non-indexed entries to zero in each iteration. Therefore, once it has finished, $(|V_1| - 1)^j \binom{k}{j}$ vectors will be created for each j. In total, $|V_1|^k = \sum_{j=0}^{k} (|V_1| - 1)^j \binom{k}{j}$. Additionally, note that for each $\mathbf{s}_i^l \in V_1^k$, we must calculate n inner products, i.e. $\mathbf{A}^l \cdot \mathbf{s}_i^l$. Therefore, generating the table $\mathcal{T}_2$ costs $|V_1|^k \cdot (n + C_1)$ operations, where C_1 is a bound on the number of operations for generating an element in V_1^k and storing it in $\mathcal{T}_2$.

Once the table is generated, the attacker then proceeds to compute guesses for $\mathbf{s}^r$. For each candidate $\mathbf{s}_j^r \in V_1^{n-k}$, the attacker computes $\mathbf{v}_j^r = \mathbf{c} - \mathbf{A}^r \cdot \mathbf{s}_j^r$, calculates $t_j^r = I(H(\mathbf{v}_j^r))$ and finally iterates through all the arrays stored at index t_j^r of $\mathcal{T}_2$. For each array $\mathbf{s}_i^l$ found, the attacker treats $\mathbf{s}_{i,j} = (\mathbf{s}_i^l | \mathbf{s}_j^r)$ as a candidate secret. The attacker then calculates the polynomial $\mathbf{f}_{i,j} = \mathbf{c}_1 - \mathbf{s}_{i,j}$ and then verifies if $\mathbf{a_q} \cdot \mathbf{f}_{i,j}$ is a valid candidate for $2\mathbf{g} + \mathbf{1}$ or not, as described in Section 5.5.2.4. By construction, if $\mathcal{T}_2$ is queried for the guess $\mathbf{v}_j^r$ and returns $\mathbf{s}_i^l$, then $MSB_b(v_{j,p}^r) = MSB_b(u_{i,p}^l)$ for all $0 \leq p \leq n_1$. If there is no candidate secret $\mathbf{s}_i^l$ found, call for more samples and repeat.

Generating all possible guesses $\mathbf{s}_j^r \in V_1^{n-k}$ is done using a similar approach as the algorithm used to enumerate all possible candidates in V_1^k. Therefore, generating an element in V_1^{n-k} has a cost of C_1 operations on average.[2] Also, note that for each candidate $\mathbf{s}_j^r$, the attacker must calculate n inner products (plus a subtraction) and calculate an index t_j^r in $\mathcal{T}_2$. Then the attacker calculates a polynomial multiplication for each array found in the linked list pointed to by $\mathcal{T}_2[t_i^l]$. It is expected that the number of entries of any linked list pointed to by any entry of $\mathcal{T}_2$ is bounded by a constant C_2. Therefore, since there are $|V_1|^{n-k}$ possible guesses $\mathbf{s}_j^r$, the overall cost of the second part of the attack is $|V_1|^{n-k} \cdot (n + C_1 + C_2)$ operations.

A drawback of the above algorithm is its memory requirements, as it requires storing $|V_1|^k$ vectors in memory. In the next section, we will describe a different algorithm to search for collisions with only negligible memory requirements. This line of research started with

[2] Assuming $k = n/2$.

Pollard's ρ-method, which was originally applied to factoring and discrete logarithms [57], but may be generalised to finding collisions in any function [70].

5.5.3.2 Parallel Collision Search

Parallel collision search is a method to search for colliding values x, y in the function values $F(x)$, $F(y)$ for a given function F. This technique can be applied to meet-in-the-middle attacks [70, 36, 71]. We will follow the description of the attack in [70].

The goal in collision search is to create an appropriate function F and find two different inputs that produce the same output. F is required to have the same domain and codomain, i.e. $F : S \to S$, and to be sufficiently complex that it behaves like a random mapping.

To perform a parallel collision search, each processor (thread) proceeds as follows: Select a starting point $x_0 \in S$ and produce the trail of points $x_i = F(x_{i-1})$, for $i = 1, 2, \ldots$ until a distinguished point x_d is reached based on some easily testable distinguishing property, e.g. a fixed number of leading zeros. For each trail we store the triples (x_0, x_d, d) in a single common list for all processors (threads) and start producing a new trail from a new starting point. Whenever we find two triples (x_0, x_t, t), (x'_0, x'_t, t') with $x_t = x'_t$ and $x_0 \neq x'_0$ we have found a collision. These trails can be re-run from their starting values to find the steps $x_i \neq x_j$ for which $F(x_i) = F(x_j)$. It can then be checked if this is the collision we were looking for.

This method can fail in one of two ways: It is possible for one trail to collide with the starting point of another trail in which case we have a "Robin Hood" which does not lead to a collision in F and can be easily detected. The second type of hazard is when a trail falls into a loop which contains no distinguished point. Left undetected, the processor (thread) involved would cease to contribute to the collision search. This problem can be handled by setting a maximum trail length, e.g. setting the maximum trail length to $\frac{20}{\theta}$ as suggested in [70], where θ is the proportion of points which satisfy the distinguishing property, and abandoning any trail which exceeds the maximum length.

In a general meet-in-the-middle attack, we have two functions, $F_1 : W_1 \to S$ and $F_2 : W_2 \to S$, and we wish to find two particular inputs $w_1 \in W_1$ and $w_2 \in W_2$, such that

$F_1(w_1) = F_2(w_2)$. To apply parallel collision search to meet-in-the-middle attacks, we construct a single function F which has identical domain and codomain to do the collision search on. This function may be constructed from the functions F_1 and F_2 as follows. Let I be the set $\{0, 1, \ldots, M-1\}$, with $M \geq \max\{|W_1|, |W_2|\})$, and let $D_1 : I \to W_1$ and $D_2 : I \to W_2$ be functions that map elements of the interval onto elements of W_1 and W_2 respectively.

Moreover, let $G : S \to I \times \{0, 1\}$ be a mapping that maps elements from the range of F_i to elements of I and a bit selector. The mapping G should distribute the elements of S fairly uniformly across $I \times \{0, 1\}$. As a good example for G, we may take a hash function of whose output the most significant bit is split off. Therefore, we can now define the function F as $F(x, i) = G(F_{i+1}(D_{i+1}(x)))$. This is a function whose domain is equal to its codomain and on which collision search may be performed.

Parallel Collision Search on LWE In order to apply the previous idea to search for $\mathbf{s}^l$ and $\mathbf{s}^r$, we have to define a function F with the same domain and codomain. We first define F_1 as $F_1(\mathbf{x}) = H(\mathbf{A}^l \cdot \mathbf{x})$ and F_2 as $F_2(\mathbf{x}) = H(\mathbf{c} - \mathbf{A}^r \cdot \mathbf{x})$, where H is the function defined in Section 5.5.3.1. The domain of F_1 is V_1^k, while the domain of F_2 is V_1^{n-k}, and both functions have the same codomain.

We next define $|I| = \max(|V_1|^k, |V_1|^{n-k})$, the function $D_1 : I \to V_1^k$ as a function mapping an integer to a vector in V_1^k and the function $D_2 : I \to V_1^{n-k}$ as a function mapping an integer to a vector in V_1^{n-k}. The problem of assigning deterministic indices to binary sequences of length n and weight d is well-known in the combinatorial literature, in particular we make use of number systems [43]. Indeed, let $\mathtt{A}$ be the array whose entries are all the elements of V_1, i.e. $\mathtt{A}[0] = -4, \ldots, \mathtt{A}[7] = 4$. The function D_1 will construct a vector $\mathbf{v} \in V_1^k$ given a $p \in I$ as follows. It first sets $p' = p \mod |V_1|^k$, then calculates $0 \leq c_i < |V_1|$ such that $p' = c_0 + c_1 \cdot |V_1| + \ldots + c_{k-1} \cdot |V_1|^{k-1}$, i.e. the representation of p' on base $|V_1|$. It then calculates $\mathbf{v} = [v_0, v_1, \ldots, v_{k-1}]$ by simply assigning the value $A[c_i]$ to v_i, and lastly returns the vector $\mathbf{v}$. D_2 may be defined in a similar manner.

Let H_g be a hash function. This can be any function with codomain larger than I. Therefore, $G(x) = (H_g(x) \mod |I|) \times MSB(H_g(x))$ and

$$\begin{aligned} F : I \times \{0,1\} &\longmapsto & I \times \{0,1\} \\ (x,i) &\longmapsto G(F_{i+1}(D_{i+1}(x))). \end{aligned}$$

The function F can now be used for a collision attack as follows: We define a distinguishing property $\mathcal{D}$ on I and create trails starting from a point $x_0 \in I$ chosen at random. We run trails until they reach a distinguished point $x_t \in \mathcal{D}$ and then store the triple (x_0, x_t, t) in a hash-list.

Whenever we find two triples (x_0, x_t, t), (x'_0, x'_t, t') with $x_t = x'_t$ and $x_0 \neq x'_0$ we have found a collision. The trails then will be re-run from their starting values to find the steps $x_i \neq x_j$ for which $F(x_i, b_i) = F(x_j, b_j)$.

We then compute $\mathbf{s}' = (\mathbf{s}^l | \mathbf{s}^r)$ with either $\mathbf{s}^l = D_{b_i+1}(x_i)$ and $\mathbf{s}^r = D_{b_j+1}(x_j)$ when $b_i = 0$ and $b_j = 1$ or $\mathbf{s}^l = D_{b_j+1}(x_j)$ and $\mathbf{s}^r = D_{b_i+1}(x_i)$ when $b_j = 0$ and $b_i = 1$. It is then checked if $\mathbf{f}' \cdot \mathbf{a_q}$ is a valid candidate for $2\mathbf{g} + \mathbf{1}$ or not, where $\mathbf{f}' = \mathbf{c}_1 - \mathbf{s}'$. If so, then return $\mathbf{f}'$ and $\mathbf{f}' \cdot \mathbf{a_q}$. If not, the triple (x_0, x_t, t) is replaced by (x'_0, x'_t, t') in the hash-list.

5.5.3.3 Hybrid attack on LWE

Another related attack idea is to have a hybrid algorithm by combining lattice-reduction and meet-in-the-middle techniques. This attack idea has been applied against NTRU in [35], as well as on special instances of the Learning with Errors (LWE) problem in [16, 72]. With notation as in Equation (5.4), to recover $\mathbf{s}$, the attacker first tries to guess $\mathbf{s}^l$ and solve the remaining LWE instance $\mathbf{c}' = \mathbf{c} - \mathbf{A}^l \cdot \mathbf{s}^l = \mathbf{A}^r \cdot \mathbf{s}^r + \mathbf{e} \mod q$. The newly obtained LWE instance may be solved by solving a close vector problem in the lattice $\mathtt{L}_q(\mathbf{A}^r)$. In more detail, $\mathbf{c}' = \mathbf{A}^r \cdot \mathbf{s}^r + q\mathbf{w} + \mathbf{e}$, for some vector $\mathbf{w} \in \mathbb{Z}^n$, is close to the lattice vector $\mathbf{A}^r \cdot \mathbf{s}^r + q\mathbf{w} \in \mathtt{L}_q(\mathbf{A}^r)$, since $\mathbf{e}$ is very likely to have many zeros and its non-zero entries are small. Hence one can hope to find $\mathbf{e}$ by running Babai's Nearest Plane algorithm in combination with a sufficient basis reduction as a pre-computation. Moreover, the attacker may speed up the guessing part of the attack by using a Meet-in-the-Middle approach, i.e. guessing vectors $\mathbf{s}^l_1 \in V_1^k$ and $\mathbf{s}^l_2 \in V_1^k$ such that $\mathbf{s}^l = \mathbf{s}^l_1 + \mathbf{s}^l_2$.

5.5.3.4 Combining Lattice Techniques and Key Enumeration

In this section we expand on a hybrid attack similar to the previously described. First we will give a general description of the attack by following the description in [72] and then show a manner of applying it to our problem of recovering the private key in the cold boot attack setting.

According to [72], the goal of this hybrid attack is to find a shortest vector in a lattice $\mathtt{L}$, given a basis of $\mathtt{L}$ of the form

$$\mathbf{B}' = \left(\begin{array}{c|c} \mathbf{B} & \mathbf{C} \\ \hline \mathbf{0} & \mathbf{I}_k \end{array}\right),$$

where $0 < k < m$, $\mathbf{B} \in \mathbb{Z}^{(m-k)\times(m-k)}$, and $\mathbf{C} \in \mathbb{Z}^{(m-k)\times k}$.

Let $\mathbf{v}$ be a short vector contained in the lattice $\mathtt{L}$. Let us split the short vector $\mathbf{v}$ into two parts $\mathbf{v} = (\mathbf{v}^l|\mathbf{v}^r)$ with $\mathbf{v}^l \in \mathbb{Z}^{m-k}$ and $\mathbf{v}^r \in \mathbb{Z}^k$. The left part $\mathbf{v}^l$ of $\mathbf{v}$ will be recovered with lattice techniques (solving BDD problems), while the right part $\mathbf{v}^r$ will be recovered by guessing during the attack. Because of the special form of the basis $\mathbf{B}'$, we have that

$$\begin{pmatrix} \mathbf{v}^l \\ \mathbf{v}^r \end{pmatrix} = \mathbf{B}' \begin{pmatrix} \mathbf{x} \\ \mathbf{v}^r \end{pmatrix} = \begin{pmatrix} \mathbf{B}\cdot\mathbf{x} + \mathbf{C}\cdot\mathbf{v}^r \\ \mathbf{v}^r \end{pmatrix},$$

for some vector $\mathbf{x} \in \mathbb{Z}^{m-k}$. Thus $\mathbf{C}\cdot\mathbf{v}^r = -\mathbf{B}\cdot\mathbf{x} + \mathbf{v}^l$. Note that $\mathbf{C}\cdot\mathbf{v}^r$ is close to the lattice $\mathtt{L}(\mathbf{B})$, since it only differs from the lattice by the short vector $\mathbf{v}^l$. Therefore $\mathbf{v}^l$ may be recovered solving a BDD problem if $\mathbf{v}^r$ is known. The idea now is that if we can correctly guess the vector $\mathbf{v}^r$, then we can hope to find $\mathbf{v}^l$ by using Babai's Nearest Plane algorithm, i.e. $\mathbf{NP}_{\mathbf{B}}(\mathbf{C}\cdot\mathbf{v}^r) = \mathbf{v}^l$ if the basis $\mathbf{B}$ is sufficiently reduced. The guessing of $\mathbf{v}^r$ is normally carried out by a meet-in-the-middle attack [35, 16, 72]. Note that the lattice $\mathtt{L}(\mathbf{B})$ in which we need to solve BDD has smaller dimension, i.e. $m-k$ instead of m. Thus, the newly obtained BDD problem is expected to be easier to solve than the original SVP instance.

Returning to our problem of recovering the private key, let us assume that $\mathbf{a_q}$ is invertible

in R_q, which is the case with very high probability. Therefore, we may obtain the equation $\mathbf{f} = (2\mathbf{g} + \mathbf{1})\mathbf{a_q}^{-1} \in R_q$. Now let us set $\mathbf{v} = (\mathbf{f}|2\mathbf{g} + \mathbf{1})$ and so

$$\begin{pmatrix} \mathbf{f} \\ \\ 2\mathbf{g} + \mathbf{1} \end{pmatrix} = \begin{pmatrix} (2\mathbf{g} + \mathbf{1})\mathbf{a_q}^{-1} + q\mathbf{w} \\ \\ 2\mathbf{g} + \mathbf{1} \end{pmatrix} = \left(\begin{array}{c|c} q\mathbf{I_n} & \mathbf{A} \\ \hline \mathbf{0} & \mathbf{I_n} \end{array}\right) \begin{pmatrix} \mathbf{w} \\ \\ 2\mathbf{g} + \mathbf{1} \end{pmatrix},$$

for some $\mathbf{w} \in \mathbb{Z}^n$, where $\mathbf{A}$ is the rotation matrix of $\mathbf{a_q^{-1}}$, i.e.

$$\mathbf{A} = \begin{pmatrix} a_0 & -a_{n-1} & \cdots & -a_1 \\ \\ a_1 & a_0 & \cdots & -a_2 \\ \\ \vdots & \vdots & \cdots & \vdots \\ \\ a_{n-1} & a_{n-2} & \cdots & a_0 \end{pmatrix},$$

and

$$q\mathbf{I}_n = \begin{pmatrix} q & 0 & \cdots & 0 \\ \\ 0 & q & \cdots & 0 \\ \\ \vdots & \vdots & \cdots & \vdots \\ \\ 0 & 0 & \cdots & q \end{pmatrix}.$$

Hence, $\mathbf{v} \in \mathtt{L}$ with

$$\mathtt{L} = \mathtt{L}\left(\left(\begin{array}{c|c} q\mathbf{I}_n & \mathbf{A} \\ \hline \mathbf{0} & \mathbf{I}_n \end{array}\right)\right).$$

Since the basis of $\mathtt{L}$ has the required form and $\mathbf{f}$ is a short vector,[3] the attacker may perform the hybrid attack to try to recover the vector $\mathbf{v}$ by performing the guessing of the vector $2\mathbf{g} + \mathbf{1}$ via running an instance of a key enumeration algorithm. Specifically, the attacker now uses the algorithm described in Section 5.5.2.2 to generate valid and high scoring candidate polynomials $\mathbf{h}_i$ for $2\mathbf{g} + \mathbf{1}$ from $\mathbf{s}_2'$. For each $\mathbf{h}_i$, the attacker calculates $\mathbf{NP}_{\mathbf{B}}(\mathbf{a_q}^{-1} \cdot \mathbf{h}_i) = \mathbf{v}^l$, where $\mathbf{B} = q\mathbf{I}_n$. If $\mathbf{f}' = \mathbf{v}^l \mod q$ is a valid candidate for $\mathbf{f}$, then the algorithm has found the pair $(\mathbf{f}', \mathbf{h}_i)$.

On the other hand, as far as performance is concerned, the attacker can perform the guessing of $2\mathbf{g} + \mathbf{1}$ in parallel by carefully selecting which key enumeration algorithm to run. As in other settings, the OKEA algorithm would not be appropriate, since it is inherently a serial algorithm and requires a lot of memory as discussed in Section 3.2. However, non-optimal key enumeration algorithms would suit better, since they can be customised to do a parallel and memory-efficient search by either selecting a suitable interval in which the outputs' scores lie, e.g. the algorithm described in Section 3.2.3, or selecting a suitable interval in which the outputs' ranks lie, e.g. the algorithm described in Section 3.2.5.

Note that this technique is rather similar to the key recovery algorithm introduced in Section 5.5.2.4 (which will be tested in Section 5.6). In the sense that both techniques perform an enumeration of candidates (one does for $\mathbf{f}$, while the other does for $2\mathbf{g} + \mathbf{1}$) and then each uses its corresponding oracle to find a valid pair $(\mathbf{f}_i, \mathbf{h}_i)$.

5.6 Experimental Evaluation

5.6.1 Simulations

All the source code of the implemented algorithms is written in Java. In particular, we make use of some implemented key enumeration algorithms described in Chapter 3. To simulate the performance of our key recovery algorithms, we generate a private key (according to the format), flip its bits according to the error probabilities α, β, and then run our chosen key recovery algorithm. We refer to such a run attempting to recover a single private key as a *simulation*. For our experiments, we ran our simulations on a machine with Intel Xeon CPU E5-2667 v2 cores running at 3.30GHz with 8 cores.

[3] This follows because of its form.

5.6.2 Key Recovery Via Key Enumeration

In this section, we will show results obtained from running the key recovery algorithm introduced in Section 5.5.2.4.

5.6.2.1 Parameters

For the simulations, we used two parameter sets: `BLISS-I` with $n = 512$, $q = 12289$, $d_1 = 154$, $d_2 = 0$ and `BLISS-IV` with $n = 512$, $q = 12289$, $d_1 = 231$, $d_2 = 31$.

5.6.2.2 Setup

Recall that we have access to $\mathbf{s}'_1$ that is a bit string of size $W = 8(512) = 4096$. We first sets a chunk to be an 8-bit string, i.e. $w = 8$, and so there are $n = 512$ chunks. When using the parameter set `BLISS-I`, we set the candidates for a chunk to be the integers in the set $\{-1, 0, 1\}$. On the other hand, when using the parameter set `BLISS-IV`, we set the candidates for a chunk to be the integers in the set $\{-2, -1, 0, 1, 2\}$. Also, we set a block to be a successive sequence of 64 chunks, resulting in 8 blocks. Besides, we set the number of candidates `blsize` generated for any block in `Phase I` to 2^r, for $r = 8, 9, 10$. Thus, eight candidate lists, each of size 2^r, will be obtained from `Phase I`. Regarding the key enumeration algorithm to be employed in `Phase II`, we will make use of a key enumeration algorithm that combines key features from the key enumeration algorithms discussed in Chapter 3. We next describe our particular choice of key enumeration algorithm.

5.6.2.3 Key Enumeration Algorithm for `Phase II`

First note that because of the nature of the log-likelihood function employed to calculate scores, many chunk candidates of any given list L_{b^i} output by any `Phase I` (by OKEA) will have a repeated score value. Taking advantage of this observation, we will design an algorithm to both efficiently count and enumerate all the key candidates that any `Phase II` will consider in any given interval $[a, b]$.

First we define a "compact" list as a 2-tuple of the form ($rscore$, `rlist`), where $rscore$ is a real value representing a repeated score, while `rlist` is a list of chunk candidates such

that each chunk candidate has the repeated score as its score. From each list L_{b^i}, with $0 \leq i < n_b$, we can obtain S_i "compact" lists $C^i_{k_i}$, with $0 \leq k_i < S_i$. We can then pick a "compact" list $C^i_{k_i}$ per list L_{b^i} and simultaneously compute the sum of scores and product of counts (i.e. the sizes of $C^i_{k_i}$.`rlist`) to produce a 3-tuple of the form (*tscore*, *tnumber*, `rtable`) and insert it into the list L_{bt}. The component *tnumber* holds the total number of key candidates that will have the total accumulated score *tscore*, while `rtable` is a table whose size is the number of blocks and whose entry i points to the list $C^i_{k_i}$. `rlist`. Note that storing the `rtable` in each 3-tuple avoids doing a "Bin Decomposition" process as Algorithm 15 from Section 3.2.5.

Since the number of "compact" lists, S_i, obtained from each list L_{b^i} is less than 10 on average in our experiments, the total number of entries of L_{bt}, $\prod_{i=0}^{n_b-1} S_i$, is not expected to be considerably large if the number of blocks, n_b, is selected suitably. Furthermore, according to our experiments, it is preferable to select the number of blocks to be less than 12 so as not to negatively affect the overall performance because of the number of entries of L_{bt}. Also, note that all the entries of the list L_{bt} may be ordered in decreasing order based on the component *tscore*. So we may obtain an efficient algorithm for counting and enumerating the key candidates in any given interval that our `Phase II` search algorithm would need to consider.

Given the interval $[a, b]$, the algorithm first calculates the set of indices `J` such that the value $tscore_j$ from any entry with index $j \in$ `J` of L_{bt} lies in the given interval and then proceeds as follows.

On the one hand, as for counting, the algorithm will iterate through the indices $j \in$ `J` while summing the value $tnumber_j$ from the entry with index j of L_{bt}. On completion, it will return the total sum. On the other hand, as for enumeration, the algorithm will iterate through the indices $j \in$ `J` while generating all possible key candidates that can be formed from the table $\texttt{rtable}_j$ obtained from the entry with index j of L_{bt}. This can be done by simply presenting the table $\texttt{rtable}_j$ as input to the function `processKF`, which is defined in Algorithm 16 from Section 3.2.5.

When enumerating, the order in which the algorithm iterates through the indices $j \in$ `J` helps in guaranteeing some quality in the order in which the key candidates will be generated. Indeed, assuming L_{bt} is ordered in decreasing order based on the component *tscore*,

the set J once calculated may be seen as an array of the form $[j_{start}, j_{start}+1, \ldots, j_{stop}]$. Therefore, the algorithm will first generate all key candidates whose accumulated score is equal to the value $tscore_{j_{start}}$ from the entry with index j_{start} of L_{bt}, followed by all key candidates whose accumulated score is equal to the value $tscore_{j_{start}+1}$ from the entry with index $j_{start}+1$ of L_{bt} and so on. Its behaviour will be similar to the manner in which Algorithm 12 from Section 3.2.4 works.

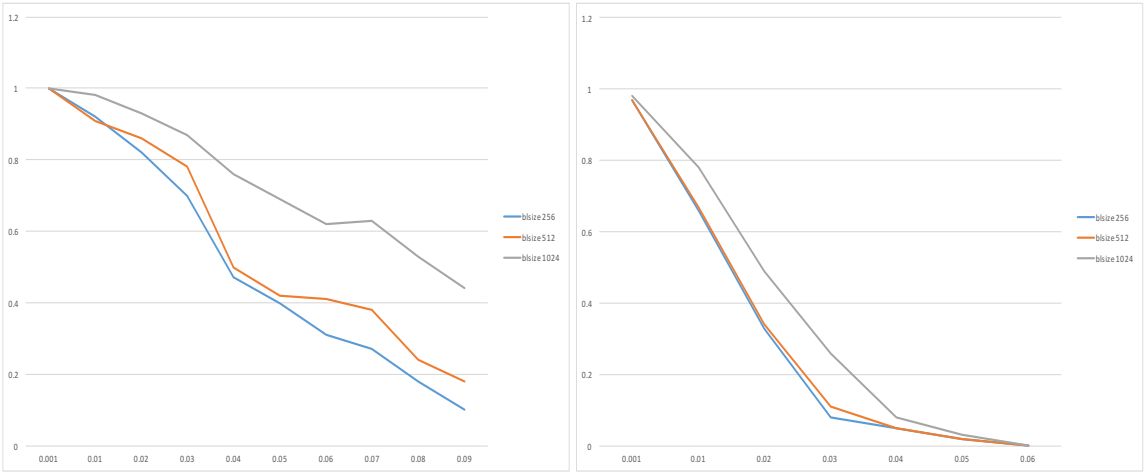
Moreover, suppose we would like to have t independent tasks $T_1, T_2, T_3, \ldots, T_t$ executing in parallel to enumerate all key candidates whose total accumulated scores are in a given interval $[B_1, B_2]$. After creating the array $\mathtt{J} = [j_{start}, j_{start}+1, \ldots, j_{stop}]$, we can partition the array J into t disjoint sub-arrays $\mathtt{J}_i$, and set each task T_i to iterate through the indices $j_i \in \mathtt{J}_i$ while generating all possible key candidates that can be formed from the table `rtable` obtained from the entry with index j_i of L_{bt}. A consequence of the independence of the tasks is that the key candidates will be generated in no particular order, i.e. this algorithm will lose its near-optimality property when running in parallel.

Recall from Section 3.2.4.2 that since we have access to the number of candidates to be enumerated, $tnumber_j$, for the score $tscore_j$ per index $j \in J$ beforehand, we may design a partition algorithm that could almost evenly distribute the workload among the tasks. The partition algorithm directly follows from the algorithm outlined in Section 3.2.5.3, viz.

1. Set i to 0.
2. If J is non-empty, pick an index j in J such that $tnumber_j$ is the maximum number. Or else return $\mathtt{J}_0, \mathtt{J}_1, \ldots, \mathtt{J}_t$.
3. Remove j from the array J and add it to the array $\mathtt{J}_i$.
4. Update i to $(i+1) \mod t$ and go back to Step 2.

5.6.2.4 Results

Complete Enumeration Here we perform simulations to estimate the expected success rate of our overall algorithm without actually executing the expensive `Phase II`. Let p_i denote the probability that the correct chunk candidate is actually found in the list L_{b^i}; p_i will be a function of r. It follows that the probability that our `Phase II` algorithm outputs the correct candidate for $\mathbf{f}$ when performing a *complete* enumeration over all 2^{8r}



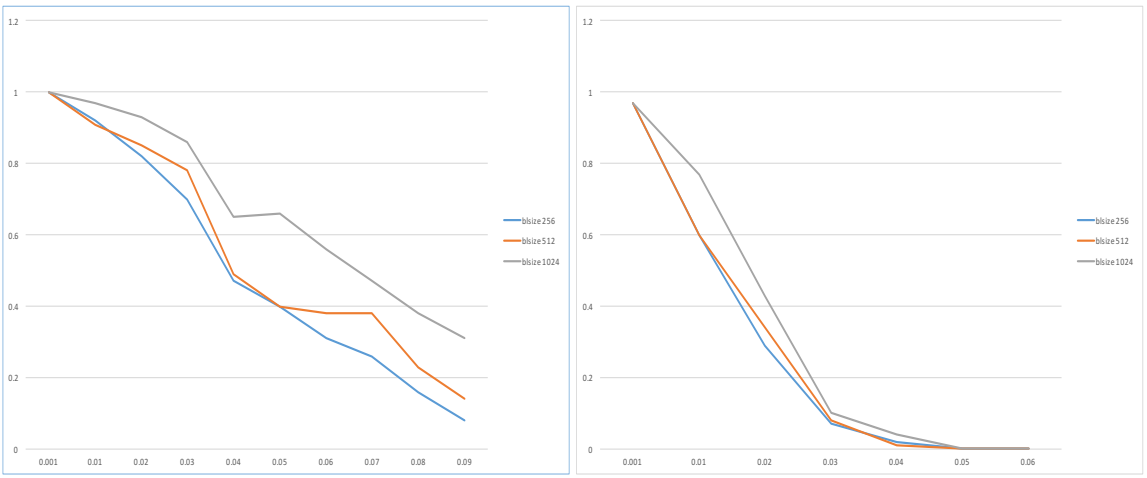
(a) `BLISS-I` parameters. (b) `BLISS-IV` parameters.

Figure 5.2: Expected success rate for a full enumeration for $\alpha = 0.001$. The y-axis represents the success rate, while the x-axis represents β.

candidate keys is given by $p = \prod_{i=1}^{8} p_i$. This simple calculation gives us a way to perform simulations to estimate the expected success rate of our overall algorithm without actually executing the expensive `Phase II`. We simply run many simulations of `Phase I` for the given value of `blsize` (each simulation generating a fresh private key and perturbing it according to α, β), and, after each simulation, test whether the correct chunks of $\mathbf{f}$ are to be found in the lists L_{b^i}.

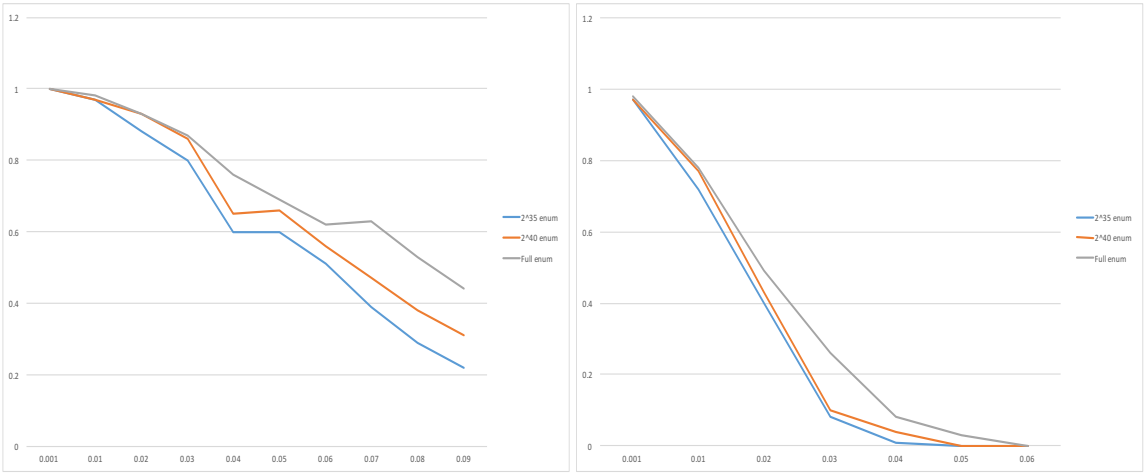
Figure 4.2 shows the success rates for complete enumeration for values of `blsize` $= 2^r$ for $r \in \{8, 9, 10\}$. As expected, the greater the value of `blsize`, the higher the success rate for a fixed α and β. Also, note that when β increases, the success rate drops to zero. This is expected since it is likely that at least one chunk of $\mathbf{f}$ will not be included in the corresponding lists coming out of `Phase I` when the noise levels are high, at which point `Phase II` inevitably fails. Additionally, note that for `BLISS-IV` parameters the success rate will drop quickly (for $\beta = 0.06$), while for `BLISS-I` parameters the success rate will drop for $\beta \geq 0.09$. This is expected because $d_2 = 0$ for the latter case and so the set of candidates for a chunk was set to $\{-1, 0, -1\}$. This case is similar to the case of NTRU described in Section 4.5.5.

Note that each data point in this figure (and all figures in section) were obtained using 100 simulations. Note that the running times for `Phase I` are very low on average (≤ 50 ms), since that phase consists of calling the OKEA for each of the eight lists with `blsize` in the set $\{256, 512, 1024\}$.



(a) `BLISS-I` parameters. (b) `BLISS-IV` parameters.

Figure 5.3: Expected success rate for a 2^{40} enumeration for $\alpha = 0.001$. The y-axis represents the success rate, while the x-axis represents β.



(a) `BLISS-I` parameters. (b) `BLISS-IV` parameters.

Figure 5.4: Expected success rate for various enumerations for $\alpha = 0.001$ and `blsize` $= 1024$. The y-axis represents the success rate, while the x-axis represents β.

Partial Enumerations Here, we only considered Class II intervals, i.e. intervals of the form $[\max - W, \max]$. To calculate a suitable value for W, we exploit the key enumeration algorithm used in `Phase II` to estimate it as a function of the total number of keys considered, K. Specifically, given a value K, we run the counting method to find a suitable index i_K of L_{bt}. Therefore, the key enumeration will run through the indices $0, 1, \ldots, i_K$. Since we can easily estimate the speed at which individual keys can be assessed, we can also use this approach to control the total running time of our algorithms.

Figure 5.3 shows how the success rate of our algorithm varies for different values of `blsize`. We observe the same trends as for full enumeration, i.e. the greater is `blsize`, the higher is

the success rate for a fixed α and β. Also, for larger values of (α, β), the success rate drops rapidly to zero. Besides, Figure 5.4 shows the success rates for a complete enumeration and partial enumerations with 2^{35} keys and 2^{40} keys. As expected, the success rate for a full enumeration is greater than for the partial enumerations (but note that a full enumeration here would require the testing of up to 2^{80} keys, which may be a prohibitive cost).

Additionally, note that for `BLISS-IV` parameters the success rate will drop quickly (for $\beta = 0.06$), while for `BLISS-I` parameters the success rate will drop for $\beta \geq 0.09$. As noted, this is expected because $d_2 = 0$ for `BLISS-I` parameters and so the set of candidates for a chunk was set to $\{-1, 0, -1\}$.

Concerning running times, we find that our code is able to test between 600 and 1000 candidates per millisecond per core during `Phase II` in our experiments.

5.7 Chapter Conclusions

In this chapter, we initiated the study of cold boot attacks on the BLISS signature scheme, a member of the same broad family of schemes that operate over polynomial rings. Our evaluation focused on an existing BLISS implementation provided by the `strongSwan` project. We first proposed a key recovery algorithm via combining key enumeration algorithms and other techniques. In particular, we made use of the general key recovery strategy developed in Chapter 4 as a core algorithm. We then established a connection between the key recovery problem in this particular case and an instance of Learning with Errors Problem (LWE). Additionally, we explored other techniques based on the meet-in-the-middle generic attack to tackle this instance of LWE and also showed a key recovery strategy combining key enumeration and lattice techniques. We then experimented with one of the introduced key recovery algorithms to explore its performance for a range of parameters and found that our key recovery algorithm was able to tolerate a noise level of $\alpha = 0.001$ and $\beta = 0.09$ for a parameter set when performing a 2^{40} enumeration. We did not run experiments for each introduced key recovery algorithm, however it may be interesting to experimentally compare all these algorithms. Alternatively, it may also be interesting to pursue the research line of developing new key recovery techniques by combining key enumeration algorithms with other techniques for solving Bounded Distance Decoding, e.g. lattice enumeration [25]. Another possible direction for future works is ex-

ploring key recovery algorithms exploiting the extra information stored in memory, such as the NTT of the coefficients of the public polynomial $(2\mathbf{g} + \mathbf{1})/\mathbf{f}$.

Chapter 6

Cold Boot Attacks on Rainbow

Contents

6.1 Introduction 126
6.2 Multivariable Cryptosystems 128
6.3 Rainbow Implementations 134
6.4 Mounting Cold Boot Attacks 141
6.5 Experimental Evaluation 152
6.6 Chapter Conclusions 157

In this chapter, we analyse the feasibility of cold boot attacks against the Rainbow signature scheme. We will review the in-memory formats for the private key of this scheme in two implementations: the `Reference` *implementation and the* `Bouncy Castle` *implementation. We will propose a key recovery strategy that exploits the structure of this signature scheme for splitting the components of the private key into chunks. By using key enumeration algorithms, we will generate high scoring candidates for each of these components and combine them to obtain high scoring candidates for the private key.*

6.1 Introduction

In this chapter, we will study the Rainbow signature scheme [19] in the cold boot attack setting. This scheme is a member of the family of asymmetric cryptographic primitives based on multivariate polynomials over a finite field K. We believe this to be the first time that this scheme is analysed in this setting, however there is a very recent study evaluating other type of side channel attacks on a variant of this signature scheme [62]. Our work is the continuation of the trend to develop cold boot attacks for different schemes as revealed

by the literature discussed at length in Section 2.2. But it is also the continuation of the evaluation of the leading post-quantum candidates against this class of attack. Such an evaluation should form a small but important part of the overall assessment of schemes in the NIST selection process for post-quantum algorithms. In particular, this scheme has been submitted as candidate to the ongoing NIST standardisation process of post-quantum signature schemes.[1]

As noted earlier, in the cold boot attack setting, any content retrieved from computer's main memory will likely be perturbed due to physical effects on the main memory after removing the power from the computer. Therefore, having knowledge about the exact formats in which the private key of the scheme is stored in memory is essential to developing key recovery attacks. The attacker's goal is to recover the original private key from a bit-flipped version of it so that it is necessary either to propose natural formats in which private key would be stored in memory or to review specific implementations of Rainbow to learn more about the formats used to store the private key. We adopt the latter approach, and we study two distinct implementations. The first, the `Reference` implementation, is written in C language and also has been included in the package submitted to the NIST process. The second is a Java implementation included in the popular `Bouncy Castle` Java crypto library. Each of these implementations stores its private keys in memory in slightly different way.

Our key recovery algorithm exploits the structure of this signature scheme, which allows us to split the private key into components. For each component, we then split it into chunks, and create log-likelihood estimates for each candidate value for each of the chunks. Each such estimate can be regarded as a per-chunk score. A log-likelihood estimate (or score) for a candidate for the complete component can then be computed by summing the per-chunk scores across the different chunks. By making use of the general key recovery strategy developed in Chapter 4 as a core algorithm, we create lists of high scoring candidates for each component and then, by using collision techniques and the structure of the public key, we verify which combination of high scoring component candidates may be the private key.

[1] See `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` for details of the NIST process.

6.2 Multivariable Cryptosystems

The existing multivariable cryptosystems can roughly be divided into explicit cryptosystems and implicit cryptosystems. Both can be used for one of two purposes: 1) encryption or 2) electronic signature [21].

Let K be a finite field. In an explicit multivariable public key cryptosystem, we have a map F from K^n to K^m such that

$$F(x_1, \ldots, x_n) = (F^{(1)}(x_1, \ldots, x_n), \ldots, F^{(m)}(x_1, \ldots, x_n)) = (y_1, \ldots, y_m),$$

where $F^{(i)}(x_1, \ldots, x_n)$ is a polynomial in $x_1, \ldots, x_n$. The key construction for this type of system is that we first build a map f from K^n to K^m such that

$$f(x_1, \ldots, x_n) = (f^{(1)}(x_1, \ldots, x_n), \ldots, f^{(m)}(x_1, \ldots, x_n)),$$

where $f^{(i)}(x_1, \ldots, x_n)$ is a polynomial in $x_1, \ldots, x_n$, and the equation $f(x_1, \ldots, x_n) = (f^{(1)}(x_1, \ldots, x_n), \ldots, f^{(m)}(x_1, \ldots, x_n)) = (a_1, \ldots, a_m)$, can be solved easily. In other words we can find a pre-image of f easily.

Then F is constructed as $F = \mathcal{L}_1 \circ f \circ \mathcal{L}_2$, where $\mathcal{L}_1$ is a randomly-chosen, invertible affine map from K^m to K^m, i.e. $\mathcal{L}_1(\mathbf{x}) = \mathbf{x} \cdot \mathbf{A}_1 + \mathbf{c}_1$, where $\mathbf{A}_1$ is an $m \times m$ invertible matrix and $\mathbf{c}_1 \in K^m$; and $\mathcal{L}_2$ is a randomly-chosen, invertible linear map from K^n to K^n, viz. $\mathcal{L}_2(\mathbf{x}) = \mathbf{x} \cdot \mathbf{A}_2 + \mathbf{c}_2$, where $\mathbf{A}_2$ is an $n \times n$ invertible matrix and $\mathbf{c}_2 \in K^n$.

In this case, the public key consists of the m polynomial components of F and the field structure of K. The secret key mainly consists of $\mathcal{L}_1$ and $\mathcal{L}_2$. The key idea is that $\mathcal{L}_1$ and $\mathcal{L}_2$ serve the purpose of 'hiding' the map f, which otherwise could be solved easily. In some systems the function f may be well-known, whereas in others f itself might be kept secret. In order to encrypt a message $\mathbf{x}'$, one calculates $F(\mathbf{x}')$. To decrypt a message $\mathbf{y}'$, one solves the equation $F(x'_1, ..., x'_n) = \mathbf{y}'$.

In the case of electronic signature, to sign a message $\mathbf{y}'$, one solves the above equation, whose solution we denote by $\mathbf{x}'$. To verify if it is a legitimate signature, one just needs to check if indeed $F(x'_1, ..., x'_n) = \mathbf{y}'$.

Due to the design, we can see that we can find a pre-image of $\mathbf{y}'$ by applying in order $(\mathcal{L}_1)^{-1}$, f^{-1} and $(\mathcal{L}_2)^{-1}$.

6.2.1 The Oil and Vinegar Signature Scheme

In this section, we will describe this scheme introduced in [56].

Definition 6.2.1 *Let us set $o, v \in \mathbb{N}$, $n = o + v$ and define an Oil-Vinegar polynomial as*

$$p(x_1, ..., x_n) = \sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{ij} \cdot x_i \cdot x_j + \sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{ij} \cdot x_i \cdot x_j + \sum_{i=1}^{n} \gamma_i \cdot x_i + \eta,$$

where $x_1, \ldots, x_v$ are called Vinegar variables, while $x_{v+1}, \ldots, x_n$ are called Oil variables.

Key Generation. Let K be a finite field and $o, v \in \mathbb{N}$. Let us set $n = o+v$. The central map $f : K^n \to K^o$ consists of o Oil-Vinegar polynomials $f^{(1)}, \ldots, f^{(o)}$, i.e.,

$$f^{(k)} = \sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{ij}^{(k)} \cdot x_i \cdot x_j + \sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{ij}^{(k)} \cdot x_i \cdot x_j + \sum_{i=1}^{n} \gamma_i^{(k)} \cdot x_i + \eta^{(k)},$$

with $\alpha_{ij}^{(k)}, \beta_{ij}^{(k)}, \gamma_i^{(k)}$ and $\eta^{(k)} \in K (1 \leq k \leq o)$.

We compose f with a randomly chosen invertible affine map $\mathcal{L}_2 : K^n \to K^n$. We then define the public key as $F = f \circ \mathcal{L}_2 : K^n \to K^o$ and the private key as f and $\mathcal{L}_2$.

Signature Generation. Given a message m, then

1. Use a hash function $H : \{0,1\}^* \to K^o$ to compute $\mathbf{y}' = H(m)$.
2. Compute a pre-image $\mathbf{x} \in K^n$ of $\mathbf{y}'$ under the central map f.
3. Compute the signature $\mathbf{x}' \in K^n$ by $\mathbf{x}' = \mathcal{L}_2^{-1}(\mathbf{x})$.

At the step 2, we need to compute a pre-image $\mathbf{x} \in K^n$ of $\mathbf{y}' = (y'_1, \ldots, y'_o)$ under the central map f. Note that the polynomials $f^{(k)}, 1 \leq k \leq o$, are of the form

$$\sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{ij}^{(k)} \cdot x_i \cdot x_j + \sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{ij}^{(k)} \cdot x_i \cdot x_j + \sum_{i=1}^{n} \gamma_i^{(k)} \cdot x_i + \eta^{(k)}.$$

Hence, if we choose the random values $r_1, \ldots, r_v$ for the Vinegar variables $x_1, \ldots, x_v$ respectively and substitute them into the polynomials $f^{(k)}$, we then have

$$y'_k = \sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{ij}^{(k)} \cdot r_i \cdot r_j + \sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{ij}^{(k)} \cdot r_i \cdot x_j + \sum_{i=1}^{v} \gamma_i^{(k)} \cdot r_i + \sum_{i=v+1}^{n} \gamma_i^{(k)} \cdot x_i + \eta^{(k)}.$$

Each equation is linear in the o Oil variables $x_{v+1}, \ldots, x_n$. Therefore, altogether we get o linear equations in the o variables $x_{v+1}, \ldots, x_n$. Hence, $x_{v+1}, \ldots, x_n$ may be recovered by Gaussian elimination. If the system has no solution, we choose other values for the Vinegar variables $x_1, \ldots, x_v$ and try again.

As an example, let us set $K = \mathbb{F}_7$, $o = v = 2$ and $f = (f^{(1)}, f^{(2)})$ with

$$f^{(1)}(\mathbf{x}) = 2x_1^2 + 3x_1x_2 + 6x_1x_3 + x_1x_4 + 4x_2^2 + 5x_2x_4 + 3x_1 + 2x_2 + 5x_3 + x_4 + 6,$$
$$f^{(2)}(\mathbf{x}) = 3x_1^2 + 6x_1x_2 + 5x_1x_4 + 3x_2^2 + 5x_2x_3 + x_2x_4 + 2x_1 + 5x_2 + 4x_3 + 2x_4 + 1.$$

The goal is to find a pre-image $\mathbf{x} = (x_1, x_2, x_3, x_4)$ of $\mathbf{y}' = (3, 4)$ under the central map F. We choose random values for x_1 and x_2, e.g., $(x_1, x_2) = (1, 4)$, and substitute them into $f^{(1)}$ and $f^{(2)}$. Hence, $f^{(1)}(1, 4, x_3, x_4) = 4x_3 + x_4 + 4$, $f^{(2)}(1, 4, x_3, x_4) = 3x_3 + 4x_4$. We now need to solve the following linear system,

$$4x_3 + x_4 + 4 = 3,$$
$$3x_3 + 4x_4 = 4.$$

The solution for this system is $(x_3, x_4) = (1, 2)$. Therefore, a pre-image of $\mathbf{y}'$ is $\mathbf{x} = (1, 4, 1, 2)$.

Signature Verification. Given a message m and a signature $\mathbf{x}' \in K^n$, then

1. Compute $\mathbf{w} = H(m)$.
2. Compute $\mathbf{w}' = F(\mathbf{x}')$.

3. Accept the signature iff $\mathbf{w}' = \mathbf{w}$.

The original scheme was broken in [41]. However, a variant of it, introduced in [40], has not been broken since 1999. This variant is called "Unbalanced Oil and Vinegar" (UOV), since it has more "Vinegar" variables than "Oil" variables. Although there is a high confidence in regard to its security, it is not the fastest multivariate scheme, since it has very large key sizes and large signatures.

6.2.2 Rainbow, a Signature Scheme

This scheme was introduced in 2005 by J. Ding and D. Schmidt [19]. It is a multi-layer version of UOV, featuring a reduced number of variables in the public key, smaller key sizes, and smaller signatures. It also is very efficient, much faster than RSA, and suitable for low-cost devices. Moreover, no weaknesses have been found on it since its introduction. Furthermore, it is a candidate in the ongoing standardisation process of post-quantum signature schemes. We will describe it by following the description of [19].

Let V be the set $\{1, 2, 3, ..., n\}$. Let $v_1, \ldots, v_u$ be u integers such that $0 < v_1 < v_2 < \cdots < v_u = n$, and define the sets of integers $V_l = \{1, 2, ..., v_l\}$ for $l = 1, ..., u$, so that we have

$$V_1 \subset V_2 \subset \cdots \subset V_u = V.$$

The number of elements in V_i is v_i. Let $o_i = v_{i+1} - v_i$, for $i = 1, \ldots, u-1$. Let O_i be the set such that $O_i = V_{i+1} - V_i$ for $i = 1, \ldots, u-1$. Let P_l be the linear space of quadratic polynomials spanned by polynomials of the form

$$\sum_{i,j \in V_l} \alpha_{i,j} x_i x_j + \sum_{i \in V_l, j \in O_l} \beta_{i,j} x_i x_j + \sum_{i \in V_{l+1}} \gamma_i x_i + \eta.$$

We can see that these are Oil and Vinegar type of polynomials such that $x_i, i \in O_l$ are the Oil variables and $x_i, i \in V_l$ are the Vinegar variables. We call $x_i, i \in O_l$ an $l-th$ layer Oil variable and $x_i, i \in V_l$ an $l-th$ layer Vinegar variable. We call any polynomial in P_l an l-th layer Oil and Vinegar polynomial. Clearly we have $P_i \subset P_j \; for \; i < j$.

Therefore, each P_l, for $l = 1, ..., u-1$, is a set of Oil and Vinegar polynomials. Each polynomial in P_l has $x_i, i \in O_l$ as its Oil variables and $x_i, i \in V_l$ as its Vinegar variables. The Oil and Vinegar polynomials in P_i can be defined as polynomials such that $x_i \in O_i$ are the Oil variables and $x_i, i \in V_i$ are the Vinegar variables. This can be illustrated by the fact that $V_{i+1} = \{V_i, O_i\}$.

Now, we will define the central map f of the Rainbow signature scheme. It is a map f from K^n to K^{n-v_1} such that

$$f(x_1, \ldots, x_n) = (f^{(1)}(x_1, \ldots, x_n), \ldots, f^{(n-v_1)}(x_1, \ldots, x_n)),$$

where the first o_1 polynomials $f^{(1)}, ..., f^{(o_1)}$ are taken from P_1, the next o_2 polynomials $f^{(o_1+1)}, ..., f^{(v_2)}$ are taken from P_2, and so on. Therefore, f consists of $u-1$ layers of Oil and Vinegar constructions. From this, we build a rainbow of our variables:

$$\begin{aligned}
&[x_1, \ldots, x_{v_1}]; \{x_{v_1+1}, \ldots, x_{v_2}\}. \\
&[x_1, \ldots, x_{v_1}, x_{v_1+1}, \cdots, x_{v_2}]; \{x_{v_2+1}, \ldots, x_{v_3}\}. \\
&[x_1, \ldots, x_{v_1}, x_{v_1+1}, \ldots, x_{v_2}, x_{v_2+1}, \ldots, x_{v_3}]; \{x_{v_3+1}, \cdots, x_{v_4}\}. \\
&\vdots \qquad\qquad ; \qquad\qquad \vdots \\
&[x_1, \ldots, \ldots, x_{v_1}, \ldots, x_{v_2}, \cdots, x_{v_3}, \ldots, \ldots, \ldots, x_{v_{u-1}}]; \{x_{v_{u-1}+1}, \cdots, x_{v_u}\}.
\end{aligned}$$

Each row above represents a layer of the rainbow. For the l-th layer above, the ones in [] are Vinegar variables, the ones in {} are Oil variables and each layer's Vinegar variables consists of all the variables in the previous layer. We call f a Rainbow polynomial map with $u-1$ layers.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two randomly chosen invertible affine maps, $\mathcal{L}_1$ is on K^{n-v_1} and $\mathcal{L}_2$ on K^n. F is defined by $F(x_1, ..., x_n) = \mathcal{L}_1 \circ f \circ \mathcal{L}_2(x_1, ..., x_n)$, which consists of $n - v_1$ quadratic polynomials with n variables.

Key Pair For the Rainbow signature scheme, the public key consists of the $n - v_1$ polynomial components of F and the field structure of K. The private key consists of the maps $\mathcal{L}_1$, $\mathcal{L}_2$ and f.

Signature Generation. Given a message m, then

1. Use a hash function $H : \{0,1\}^* \to K^{n-v_1}$ to compute $\mathbf{y} = H(m) \in K^{n-v_1}$.
2. Compute $\bar{\mathbf{y}} = \mathcal{L}_1^{-1}(\mathbf{y}) \in K^{n-v_1}$.
3. Compute a pre-image $\mathbf{x} \in K^n$ of $\bar{\mathbf{y}}$ under the central map f.
4. Compute the signature $\mathbf{x}' \in K^n$ by $\mathbf{x}' = \mathcal{L}_2^{-1}(\mathbf{x})$.

At the third step of the signature generation algorithm, we need to invert f. Hence, we need to solve the equation

$$f(x_1, \ldots, x_n) = \bar{\mathbf{y}} = (\bar{y}_1, \ldots, \bar{y}_{n-v_1}).$$

We first randomly choose the values $r_1, \ldots, r_{v_1}$ for the variables $x_1, ..., x_{v_1}$ and plug them into the first layer of o_1 equations given by

$$\begin{aligned} f^{(1)}(r_1, \ldots, r_{v_1}, \ldots, x_n) &= \bar{y}_1, \\ \vdots \qquad\qquad &= \vdots \\ f^{(o_1)}(r_1, \ldots, r_{v_1}, \ldots, x_n) &= \bar{y}_{o_1}. \end{aligned}$$

This produces a set of o_1 linear equations with o_1 variables, $x_{v_1+1}, \ldots, x_{v_2}$, which we solve to find the values of $x_{v_1+1}, \cdots, x_{v_2}$. Then we have all the values of $x_i, i \in V_2$. Then we plug these values into the second layer of polynomials, which will again produce o_2 number of linear equations, which we then solve to find the values of all $x_i, i \in V_3$. We repeat the procedure until we find a solution. If at any time, a set of linear equations does not have a solution, we will start from the beginning again by choosing another set of values for $x_1, \ldots, x_{v_1}$. We will continue until we find a solution.

Signature Verification. Given a message m, a signature $\mathbf{x}' \in K^n$, then

1. Compute $\mathbf{w} = H(m)$.
2. Compute $\mathbf{w_0} = F(\mathbf{x}')$.
3. Accept the signature $\mathbf{x}'$ if and only if $\mathbf{w_0} = \mathbf{w}$.

The NIST submission suggests using parameter sets of the form (K, v_1, o_1, o_2). In particular, its authors propose nine parameter sets for Rainbow in [20], e.g. $(\mathbb{F}_{256}, 40, 24, 24)$.

In Section 6.3.1, we expand on all the proposed parameters to use in this scheme.

6.3 Rainbow Implementations

In this section, we will review two implementations. The first is the `Reference` implementation by Jintai Ding *et al.*, while the second is the implementation provided by the `Bouncy Castle` crypto package.

6.3.1 The `Reference` Implementation

This is a reference implementation by Jintai Ding *et al.*, which has been written in the C language. This implementation has also been included in the package submitted to the NIST process.[2] In this section, we will review it, focusing on the in-memory format this particular implementation makes use of to store a rainbow private key.

Firstly we will discuss how this implementation stores elements of a finite field K, considering that K may be $\mathbb{F}_{16}$, $\mathbb{F}_{256}$, $\mathbb{F}_{31}$.

1. Elements of $\mathbb{F}_{16}$ are stored in 4 bits as linear polynomials over $\mathbb{F}_4$. The constant term of the polynomial is therefore stored in the 2 least significant bits. Elements of $\mathbb{F}_4$ are stored in two bits as linear polynomials over $\mathbb{F}_2$. The constant term of the polynomial is therefore stored in the least significant bit. Two adjacent $\mathbb{F}_{16}$ elements are packed into a single byte. Therefore, a sequence of $\mathbb{F}_{16}$ elements, of size N_s, is converted into a bit-string with size $4 \cdot N_s$. This bit-string is therefore stored as a `byte` array with $\lceil N_s/2 \rceil$ entries.
2. Elements of $\mathbb{F}_{256}$ are stored in a byte as linear polynomials over $\mathbb{F}_{16}$. The constant term of the polynomial is therefore stored in the 4 least significant bits. Therefore, a sequence of $\mathbb{F}_{256}$ elements, with size N_s, is converted into a bit-string with size $8 \cdot N_s$. This bit-string is hence stored as a `byte` array with N_s entries.
3. Elements of $\mathbb{F}_{31}$ are stored as integers in the range $[0, 30]$. Any number out of this range, for example 41, is considered as a format error. In order to convert a sequence of $\mathbb{F}_{31}$ elements, with size N_s, into a bit-string, this implementation stores 3 of elements $\mathbb{F}_{31}$ in two bytes. In case the size of such sequence, N_s, is not divisible

[2] See `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions`.

by three, this implementation stores each of the last $N_s \mod 3$ elements into 8 bits. Hence, the length of the bit-string needed to store this sequence can be calculated as $(N_s \text{ div } 3) \cdot 16 + (N_s \mod 3) \cdot 8$. This bit-string is therefore stored as a `byte` array with $2 \cdot (N_s \text{ div } 3) + (N_s \mod 3)$ entries.

Secondly we will discuss how this implementation stores each component of a private key, namely $\mathcal{L}_1$, $\mathcal{L}_2$, f. The affine map $\mathcal{L}_1 : K^{n-v_1} \to K^{n-v_1}$ will be represented as a sequence of the form

$$a^{(1)}_{1,1}, a^{(1)}_{2,1}, \ldots, a^{(1)}_{(n-v_1),1}, a^{(1)}_{1,2}, \ldots, a^{(1)}_{(n-v_1),(n-v_1)}, c^{(1)}_1, \ldots, c^{(1)}_{(n-v_1)}.$$

Therefore, the number of field elements in the sequence representing $\mathcal{L}_1$ will be $(n - v_1) \cdot (n - v_1 + 1)$. In memory, this sequence will be stored in two consecutive `byte` arrays, $\texttt{mat}_s$ and $\texttt{vec}_s$. The elements $a^{(1)}_{1,1}, a^{(1)}_{2,1}, \ldots, a^{(1)}_{(n-v_1),1}, a^{(1)}_{1,2}, \ldots, a^{(1)}_{(n-v_1),(n-v_1)}$ will be stored in the `byte` array $\texttt{mat}_s$, while the remaining elements $c^{(1)}_1, \ldots, c^{(1)}_{(n-v_1)}$ will be stored in the `byte` array $\texttt{vec}_s$. The size of each `byte` array depends on the underlying field K.

Likewise, the affine map $\mathcal{L}_2 : K^n \to K^n$ will be represented as a sequence of the form

$$a^{(2)}_{1,1}, a^{(2)}_{2,1}, \ldots, a^{(2)}_{n,1}, a^{(2)}_{12}, \ldots, a^{(2)}_{n,n}, c^{(2)}_1, \ldots, c^{(2)}_n.$$

So the number of field elements in the sequence representing $\mathcal{L}_2$ will be $n \cdot (n + 1)$. In memory, this sequence will be stored in two consecutive `byte` arrays, $\texttt{mat}_t$ and $\texttt{vec}_t$. The elements $a^{(2)}_{1,1}, a^{(2)}_{2,1}, \ldots, a^{(2)}_{n,1}, a^{(1)}_{1,2}, \ldots, a^{(1)}_{n,n}$ will be stored in the `byte` array $\texttt{mat}_t$, while the remaining elements $c^{(1)}_1, \ldots, c^{(1)}_n$ will be stored in the `byte` array $\texttt{vec}_t$. The size of each `byte` array depends on the underlying field K.

Regarding the central map, f consists of two layers, because this implementation only works using parameter sets of the form (K, v_1, o_1, o_2). Therefore,

$$f(x_1, \ldots, x_n) = (f^{(1)}(x_1, \ldots, x_n), \ldots, f^{(n-v_1)}(x_1, \ldots, x_n)),$$

with

$$f^{(k)} = \sum_{i,j,i\leq j\in V_l} \alpha_{i,j}^{(k)} x_i x_j + \sum_{i\in V_l, j\in O_l} \beta_{i,j}^{(k)} x_i x_j + \sum_{i\in V_{l+1}} \gamma_i^{(k)} x_i + \eta^{(k)},$$

for $k \in \{1, 2, \ldots, n - v_1\}$ and $l \in \{1, 2\}$. For the first layer, we have $V_1 := \{1, \ldots, v_1\}$, $O_1 := \{v_1 + 1, \ldots, v_1 + o_1\}$ and $f^{(1)}, \ldots, f^{(o_1)} \in P_1$. However, for the second layer, $V_2 := \{1, \ldots, v_2 = v_1 + o_1\}$, $O_2 := \{v_2 + 1, \ldots, n = v_2 + o_2\}$ and $f^{(o_1+1)}, \ldots, f^{(n-v_1)} \in P_2$.

Each layer of the central map f will be stored separately. In particular, let us consider the first layer. This layer consists of the polynomials $f^{(1)}, \ldots, f^{(o_1)}$. These polynomials will be divided into 3 parts, denoted as 'vv', 'vo' , and 'o-linear'.

The 'vv' part is given by

$$vv^{(k)} = \sum_{i,j,i\leq j\in V_1} \alpha_{i,j}^{(k)} x_i x_j + \sum_{i\in V_1} \gamma_i^{(k)} x_i + \eta^{(k)}, \ for\ k = 1, 2, \ldots, o_1.$$

This is a system of o_1 multivariate quadratic polynomials in v_1 variables, viz.

$$\begin{aligned}
vv^{(1)} =& \alpha_{1,1}^{(1)} x_1 x_1 + \alpha_{2,1}^{(1)} x_2 x_1 + \alpha_{2,2}^{(1)} x_2 x_2 + \alpha_{3,1}^{(1)} x_3 x_1 + \alpha_{3,2}^{(1)} x_3 x_2 + \ldots \\
& + \gamma_1^{(1)} x_1 + \gamma_2^{(1)} x_1 + \ldots + \gamma_{v_1}^{(1)} x_{v_1} + \eta^{(1)}. \\
vv^{(2)} =& \alpha_{1,1}^{(2)} x_1 x_1 + \alpha_{2,1}^{(2)} x_2 x_1 + \alpha_{2,2}^{(2)} x_2 x_2 + \alpha_{3,1}^{(2)} x_3 x_1 + \alpha_{3,2}^{(2)} x_3 x_2 + \ldots \\
& + \gamma_1^{(2)} x_1 + \gamma_2^{(2)} x_2 + \ldots + \gamma_{v_1}^{(2)} x_{v_1} + \eta^{(2)}. \\
\vdots & \qquad\qquad \vdots \\
vv^{(o_1)} =& \alpha_{1,1}^{(o_1)} x_1 x_1 + \alpha_{2,1}^{(o_1)} x_2 x_1 + \alpha_{2,2}^{(2)} x_2 x_2 + \alpha_{3,1}^{(o_1)} x_3 x_1 + \alpha_{3,2}^{(o_1)} x_3 x_2 + \ldots \\
& + \gamma_1^{(o_1)} x_1 + \gamma_2^{(o_1)} x_2 + \ldots + \gamma_{v_1}^{(o_1)} x_{v_1} + \eta^{(o_1)}.
\end{aligned}$$

So this 'vv' part will be represented as a sequence of the form

$$\gamma_1^{(1)}, \ldots, \gamma_1^{(o_1)}, \gamma_2^{(1)}, \ldots, \gamma_2^{(o_1)}, \ldots, \alpha_{1,1}^{(1)}, \ldots, \alpha_{1,1}^{(o_1)}, \alpha_{2,1}^{(1)}, \ldots, \alpha_{v_1,v_1}^{(o_1)}, \eta^{(1)}, \ldots, \eta^{(o_1)}.$$

Therefore, the number of field elements in this sequence will be $o_1(\frac{v_1(v_1+1)}{2} + v_1 + 1)$. In memory, this sequence of field elements will be stored in the `byte` array $\texttt{l1}_{vv}$.

Moreover, the 'vo' part contains the remaining quadratic terms, viz.

$$vo^{(k)} = \sum_{i \in V_1} \sum_{j \in O_1} \beta_{i,j}^{(k)} x_i x_j, \ for\ k = 1, 2, \ldots, o_1.$$

And so this part will be represented as a sequence of the form

$$\beta_{1,v_1+1}^{(1)}, \ldots, \beta_{1,v_1+o_1}^{(1)}, \beta_{2,v_1+1}^{(1)}, \ldots, \beta_{2,v_1+o_1}^{(1)}, \beta_{3,v_1+o_1}^{(1)}, \ldots, \beta_{v_1,v_1+o_1}^{(o_1)}.$$

Hence, this sequence will have $o_1^2 \cdot v_1$ elements. In memory, this sequence of field elements will be stored in the `byte` array $\texttt{l1}_{vo}$.

Finally, the 'o-linear' part contains the remaining linear terms, i.e.,

$$ol^{(k)} = \sum_{i \in O_1} \gamma_i^{(k)} x_i, \ for\ k = 1, 2, \ldots, o_1.$$

So this part will be represented as a sequence of the form

$$\gamma_{v_1+1}^{(1)}, \ldots, \gamma_{v_1+o_1}^{(1)}, \gamma_{v_1+1}^{(2)}, \ldots, \gamma_{v_1+o_1}^{(2)}, \gamma_{v_1+1}^{(3)}, \ldots, \gamma_{v_1+o_1}^{(o_1)}.$$

Therefore, the number of field elements in this sequence will be o_1^2. In memory, this sequence of field elements will be stored in the `byte` array $\texttt{l1}_{o}$.

The size of each `byte` array depends on the underlying field K. Also, these `byte` arrays will be stored in memory in the following order: $\texttt{l1}_{o}$, followed by $\texttt{l1}_{vo}$ and followed by $\texttt{l1}_{vv}$. Layer 2 is also stored in the same format. The reason for making use of this particular format for storing a layer is to improve the performance of sub-algorithms executed in the signing and verification algorithms.

Rainbow Key Pair Generation. During the execution of the key and signature generation algorithms, this implementation makes use of a large number of random field elements. These are obtained by calling a cryptographic random number generator from the OpenSSL library. The `AES CTR_DRBG` function is used as the random number generator in this implementation.

Parameter Sets. As was pointed out, this implementation works using parameter sets of the form (K, v_1, o_1, o_2). In particular, its authors propose nine parameter sets for Rainbow.

1. $(\mathbb{F}_{16}, 32, 32, 32)$, $(\mathbb{F}_{16}, 56, 48, 48)$, $(\mathbb{F}_{16}, 76, 64, 64)$.
2. $(\mathbb{F}_{31}, 36, 28, 28)$, $(\mathbb{F}_{31}, 64, 32, 48)$, $(\mathbb{F}_{31}, 84, 56, 56)$.
3. $(\mathbb{F}_{256}, 40, 24, 24)$, $(\mathbb{F}_{256}, 68, 36, 36)$,$(\mathbb{F}_{256}, 92, 48, 48)$.

Additionally, the authors of this implementation provided a security analysis of their proposed parameter sets in [20]. This analysis categorised the proposed parameter sets in various security categories specified by NIST [76]. Particularly, the parameter sets $(\mathbb{F}_{16}, 32, 32, 32)$, $(\mathbb{F}_{31}, 36, 28, 28)$ and $(\mathbb{F}_{256}, 40, 24, 24)$ are categorised in the security category I, meaning that the computational resources required to break it is comparable to or greater than the computational resources required for key search on a block cipher with a 128-bit key [76].

6.3.2 The `Bouncy Castle` Implementation

This is an implementation that has been included in the `Bouncy Castle` crypto package. This package consists of Java implementations of cryptographic algorithms. This package is organised so that it contains a light-weight API suitable for use in any environment (including the Java 2 Platform, Micro Edition) with the additional infrastructure to conform with the Java Cryptography Extension framework. It also offers the BCPQ (stands for `Bouncy Castle` Post Quantum) provider with support for the Rainbow signature algorithm.

Regarding the underlying field K, this implementation only works using $\mathbb{F}_{256}$ elements. A field element is a polynomial over $\mathbb{F}_2$ with degree at most 7. Therefore, it is represented as an 8 bit-string and is stored in the least 8 significant bits of a `short` variable (16 bits). Besides, the class `GF2Field` provides the basic operations like addition, multiplication and finding the multiplicative inverse of an element in K. These operations are implemented using the irreducible polynomial $x^8 + x^6 + x^3 + x^2 + 1$, which is represented as $101001101 =$ `0x14d`.

Concerning the in-memory representation, this implementation stores the private key as

an object of the class `RainbowPrivateKeyParameters`. This class contains the following members:

- The $(n - v_1) \times (n - v_1)$ `short` matrix `A1inv` stores the inverse of `A1`, where `A1` is the matrix of the affine map $\mathcal{L}_1$.
- The `short` array `b1`, whose size is $(n - v_1)$, stores the translation vector $\mathbf{c}_1$ of the affine map $\mathcal{L}_1$.
- The $n \times n$ `short` matrix `A2inv` stores the inverse of `A2`, where `A2` is the matrix of the affine map $\mathcal{L}_2$.
- The `short` array `b2`, whose size is n, stores the translation vector $\mathbf{c}_2$ of the affine map $\mathcal{L}_2$.
- The `int` array `vi` holds the number of Vinegar variables per layer.
- The `Layer` array `layers`. Each entry of this array stores a `Layer` object, where a `Layer` object stores the polynomials of the central map f for a layer, i.e. it represents a layer of the rainbow.

The entry l holds an object of the class `Layer` that contains the following members:

- The `int` variable `vi` holds the number of Vinegar variables for this layer v_l.
- The `int` variable `viNext` holds the number of Vinegar variables for the next layer v_{l+1}. It is equal to (the number of Oils) + (the number of Vinegars).
- The `int` variable `oi` holds the number of Oil variables in this layer o_l.
- The $o_l \times v_l \times v_l$ `short` matrix `coeff_alpha` holds the values $\alpha_{ij}^{(k)}$ for this layer. A layer has o_l polynomials, each of which has v_l^2 α-values.
- The $o_l \times o_l \times v_l$ `short` matrix `coeff_beta` holds the values $\beta_{ij}^{(k)}$ for this layer. A layer has o_l polynomials, each of which has $o_l v_l$ β-values.
- The $o_l \times v_{l+1}$ `short` matrix `coeff_gamma` holds the values $\gamma_i^{(k)}$ for this layer. A layer has o_l polynomials, each of which has v_{l+1} γ-values.
- The o_l `short` array `coeff_eta` holds the values $\eta^{(k)}$ for this layer. A layer has o_l polynomials, each of which has an η-value.

Rainbow Key Pair Generation. The class `RainbowKeyPairGenerator` contains the method `genKeyPair` that generates the private key and the public key. In particular, the private key is generated as follows:

- The affine map $\mathcal{L}_1$ is created by generating a random $(n-v_1)\times(n-v_1)$ `short` matrix `A1` and testing whether `A1` is invertible. If so, its inverse, the $(n-v_1)\times(n-v_1)$ `short` matrix `A1inv`, is calculated and then each entry of the `short` array `b1` is randomly generated.
- The affine map $\mathcal{L}_2$ is created by generating a random $n \times n$ `short` matrix `A2` and testing whether `A2` is invertible. If so, its inverse, the $n \times n$ `short` matrix `A2inv`, is calculated and then each entry of the `short` array `b2` is randomly generated.
- The `Layer` array `layers` is created by instancing each entry l as a `Layer` object for $0 \leq l <$ `numOfLayers`. Given an index l, the entry l will hold a `Layer` object that will represent the layer l and hold o_l polynomials. The instantiation of this entry l is done by
  1. Generating each `short` entry $\texttt{coeff\_alpha}^{(l)}_{(k,i,j)}$ randomly for $0 \leq k < o_l, 0 \leq i, j < v_l$.
  2. Generating each `short` entry $\texttt{coeff\_beta}^{(l)}_{(k,i,j)}$ randomly for $0 \leq k < o_l, 0 \leq i < o_l, 0 \leq j < v_l$.
  3. Generating each `short` entry $\texttt{coeff\_gamma}^{(l)}_{(k,i)}$ randomly for $0 \leq k < o_l, 0 \leq i < v_{l+1}$.
  4. Generating each `short` entry $\texttt{coeff\_eta}^{(l)}_{k}$ randomly for $0 \leq k < o_l$.

Parameter Sets. The class `RainbowParameters` represents the parameters for Rainbow. It contains the `int` array `vi`, which contains the number of Vinegar variables per layer. By default, this class fixes the set $\{6, 12, 17, 22, 33\}$, meaning that

$$v_1 = 6 \; ; o_1 = 12 - 6 = 6,$$
$$v_2 = 12; o_2 = 17 - 12 = 5,$$
$$v_3 = 17; o_3 = 22 - 17 = 5,$$
$$v_4 = 22; o_4 = 33 - 22 = 11,$$
$$v_5 = 33; o_5 = 0.$$

This parameter set was introduced by Rainbow's designers in [19]. Also, they provided a security analysis of this parameter set, which is based on examining several known attack techniques against it methodically and in detail.

6.4 Mounting Cold Boot Attacks

In this section we present our cold boot key recovery attacks on the implementations and corresponding private key formats introduced in the previous section. We continue to make the assumptions outlined in Section 2.3 and further assume that all relevant public parameters and private key formatting information are known by the adversary.

In particular, with regard to the `Reference` implementation, we assume the attacker knows the parameter set being used and obtains a noisy image of the memory that contains:

- The two consecutive `byte` arrays representing $\mathcal{L}_2$, namely $\texttt{mat}_t$ and $\texttt{vec}_t$.
- The two consecutive `byte` arrays representing $\mathcal{L}_1$, namely $\texttt{mat}_s$ and $\texttt{vec}_s$.
- The three consecutive `byte` arrays representing the layer 1, namely $\texttt{l1}_o$, $\texttt{l1}_{vo}$ and $\texttt{l1}_{vv}$.
- The three consecutive `byte` arrays representing the layer 2, namely $\texttt{l2}_o$, $\texttt{l2}_{vo}$ and $\texttt{l2}_{vv}$.

On the other hand, for the `Bouncy Castle` implementation, we assume the attacker knows the parameter set being used and obtains a noisy image of the memory that contains:

- The two consecutive `short` arrays representing $\mathcal{L}_1$, namely `A1inv` and `b1`.
- The two consecutive `short` arrays representing $\mathcal{L}_2$, namely `A2inv` and `b2`.
- The four consecutive `short` arrays representing the layer 1, namely $\texttt{coeff\_alpha}^{(1)}$, $\texttt{coeff\_beta}^{(1)}$, $\texttt{coeff\_gamma}^{(1)}$ and $\texttt{coeff\_eta}^{(1)}$. These are followed by the four consecutive `short` arrays representing the layer 2, namely $\texttt{coeff\_alpha}^{(2)}$, $\texttt{coeff\_beta}^{(2)}$, $\texttt{coeff\_gamma}^{(2)}$ and $\texttt{coeff\_eta}^{(2)}$. These arrays are then followed by other consecutive arrays representing the next layer and subsequent layers in the same order.

6.4.1 Key Recovery Algorithm

In this section, we will introduce a key recovery algorithm for the rainbow signature scheme in the cold boot attack setting. This algorithm will exploit the structure of the public function F.

We further assume the adversary has access to a message m and its corresponding sig-

nature $\mathbf{x}' = (x'_1, ..., x'_n) \in K^n$. Because the attacker knows the public key F and knows $F(x'_1, ..., x'_n) = \mathcal{L}_1 \circ f \circ \mathcal{L}_2(x'_1, ..., x'_n) = H(m)$, the natural approach the attacker may try to perform is the following:

1. The attacker constructs a set of high scoring candidates for $\mathcal{L}_1$, $C^{(\mathcal{L}_1)}$.
2. The attacker constructs a set of high scoring candidates for $\mathcal{L}_2$, $C^{(\mathcal{L}_2)}$.
3. The attacker constructs a set of high scoring candidates for f, $C^{(f)}$.
4. For each $h_i \in C^{(\mathcal{L}_1)}$, each $g_j \in C^{(\mathcal{L}_2)}$ and each $f_k \in C^{(f)}$, the attacker generates candidates $F' = h_i \circ f_k \circ g_j$ for F and verifies if $F'(x'_1, ..., x'_n) = H(m)$. If so, then the 3-tuple (h_i, g_j, f_k) is a candidate for the private key.

A potential problem with this approach is that the fourth step may be computationally costly, because the cost of this step is $\mathcal{O}(|C^{(\mathcal{L}_1)}| \cdot |C^{(\mathcal{L}_2)}| \cdot |C^{(f)}|)$. So as to reduce the computational cost that the fourth step may incur, we will make use of a time-memory trade-off, introducing two extra tables $\mathcal{S}$ and $\mathcal{T}$.

On the one hand, the table $\mathcal{S}$ will be a hash table to store 2-tuples of the form $(\mathbf{y}, i)$, where $\mathbf{y} \in K^{n-v_1}$ and i is a positive integer that represents an index. We store such a tuple $(\mathbf{y}, i)$ in this hash table $\mathcal{S}$ as follows. We first calculate a hash value $k_{y,i}$ from all the entries of the array representation $\mathtt{y}$ of $\mathbf{y}$. We then calculate an index $t_{y,i}$ from $k_{y,i}$ using a map I and finally add the tuple $(\mathbf{y}, i)$ as the first entry of a linked list pointed to by the entry $\mathcal{S}[t_{y,i}]$. As has been pointed out, the number of operations for both adding a tuple to the hash table $\mathcal{S}$ and looking an element up in this table are expected to be constant on average.

On the other hand, the table $\mathcal{T}$ will be employed to store 2-tuples of the form $(\mathbf{y}', j)$, where $\mathbf{y}' \in K^n$ and j is a positive integer that represents an index. Moreover, we will exploit the fact that $\mathcal{L}_1$ has an easily computable inverse map $\mathcal{L}_1^{-1}$ and that

$$\mathcal{L}_1 \circ f \circ \mathcal{L}_2(x'_1, ..., x'_n) = H(m) \iff f \circ \mathcal{L}_2(x'_1, ..., x'_n) = \mathcal{L}_1^{-1}(H(m)).$$

Therefore, our key recovery algorithm may be refined and now proceeds at a high level as follows:

1. The attacker constructs a list of high scoring candidates for $\mathcal{L}_1$, $C^{(\mathcal{L}_1)}$.

2. The attacker constructs a list of high scoring candidates for $\mathcal{L}_2$, $C^{(\mathcal{L}_2)}$.
3. The attacker constructs a list of high scoring candidates for f, $C^{(f)}$.
4. For each $h_i \in C^{(\mathcal{L}_1)}$, the attacker computes $\mathbf{y} = h_i^{-1}(H(m))$ and stores $\mathbf{y}$ and i in the table $\mathcal{S}$.
5. For each $g_j \in C^{(\mathcal{L}_2)}$, the attacker computes $\mathbf{y}' = g_j(\mathbf{x}')$ and stores $\mathbf{y}'$ and j in the table $\mathcal{T}$.
6. For each $f_k \in C^{(f)}$ and each $(\mathbf{y}', j) \in \mathcal{T}$, the attacker computes $\mathbf{r} = f_k(\mathbf{y}')$ and looks $\mathbf{r}$ up in the table $\mathcal{S}$. If a match is found, then $\mathbf{r} = f_k(g_j(\mathbf{x}')) \in \mathcal{S}$, so $f_k(g_j(\mathbf{x}')) = h_i^{-1}(H(m))$, for some i. Hence, the 3-tuple (h_i, g_j, f_k) is a candidate for the private key.

Note that in this variant the sixth step may incur a high cost, however this cost is now just $\mathcal{O}(|C^{(\mathcal{L}_2)}| \cdot |C^{(f)}|)$. We will next expand more on the first three steps of this variant and then on its running time, its memory consumption and its parallelisation.

6.4.1.1 Constructing High Scoring Candidates for an Affine Map

Here we will present algorithms to construct a list of high scoring candidates for an affine map $\mathcal{L} : K^n \to K^n$. These algorithms may be executed to carry out the first two steps of the key recovery algorithm introduced in Section 6.4.1.

In the `Reference` implementation, an affine map $\mathcal{L}$ is represented by two consecutive `byte` arrays, namely `mat` and `vec`. Recall that the attacker gets noisy versions of these two `byte` arrays. Let us name them as `mat`$'$ and `vec`$'$. In order to construct high scoring candidates for $\mathcal{L}$, the attacker makes use of the key recovery technique introduced in Section 4.4.1 as follows.

The attacker first regards both `mat`$'$ and `vec`$'$ as a single `byte` array `r`. This array `r` has n_r entries, where n_r depends on the underlying field K. Indeed, if $K = \mathbb{F}_{16}$, then $n_r = \lceil (n^2 + n)/2 \rceil$. If $K = \mathbb{F}_{31}$, then $n_r = 2 \cdot ((n^2 + n) \text{ div } 3) + (n^2 + n) \mod 3$. And if $K = \mathbb{F}_{256}$, then $n_r = n^2 + n$.

The attacker now regards a `byte` entry as a chunk. Therefore, $W = 8 \cdot n_r$ and $w = 8$. Concerning the candidates a chunk may have, we analyse three cases:

1. When $K = \mathbb{F}_{16}$, the candidates for any chunk may be seen as all the integers in the set $\{0, 1, 2, \ldots, 255\}$ if $n^2 + n$ is even. Otherwise, if $n^2 + n$ is odd, the candidates for any chunk save the last chunk may be seen as all the integers in the set $\{0, 1, 2, \ldots, 255\}$, while the candidates for the last chunk may be seen as all the integers in the set $\{0, 16, 32, 48, \ldots, 240\}$. The reason for this is that the last element is stored in the four most significant bit of the last `byte` entry. So the four least significant bits of the entry are zeros.
2. When $K = \mathbb{F}_{31}$, let us consider the first $2 \cdot ((n^2 + n) \text{ div } 3)$ chunks. Recall that each three elements are packed in two consecutive bytes and the least significant bit is not used.[3] So we consider the chunks $\mathtt{r}^{2k}$ and $\mathtt{r}^{2k+1}$, for $0 \leq k < (n^2 + n) \text{ div } 3$. On the one hand, the candidates for the chunk $\mathtt{r}^{2k}$ may be seen as all the integers in the set $\{0, 1, 2, \ldots, 247\}$. The reason for this is that the five most significants bits may assume any binary string of length 5 except for 11111 and the three least significant bits may assume any binary string of length 3. On the other hand, the candidates for the chunk $\mathtt{r}^{2k+1}$ may be seen as all the integers in the set $\{0, 2, 4, \ldots, 60\} \cup \{64, 66, \ldots, 124\} \cup \{128, 130, \ldots, 188\} \cup \{192, 194, \ldots, 254\}$. The reason for this is that the two most significant bit may assume any binary string of length 2, the next five consecutive bits may assume any binary string of length 5 except for 11111, and the least significant bit is 0. Concerning the remaining $(n^2 + n) \mod 3$ chunks, their candidates may be seen as all the integers in the set $\{0, 1, 2 \ldots, 30\}$.
3. When $K = \mathbb{F}_{256}$, the candidates for any chunk may be seen as all the integers in the set $\{0, 1, 2, \ldots, 255\}$.

Furthermore, the attacker represents $\mathtt{r}$ as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j}, with $n_{b^j} > 0$, consecutive chunks, such that $n_r = \sum_{j=0}^{n_b - 1} n_{b^j}$. Therefore,

$$\mathtt{r} = \mathtt{b}^0 || \mathtt{b}^1 || \cdots || \mathtt{b}^{n_b - 1},$$

where

$$\mathtt{b}^j = \mathtt{r}^{i_j} || \mathtt{r}^{i_j + 1} || \cdots || \mathtt{r}^{i_j + n_{b^j} - 1},$$

for $0 \leq j < n_b$ and some $0 \leq i_j < n_r$. The attacker now proceeds as follows.

`Phase I`

[3] See function `gf31_quick_unpack` in the file `gf31_convert.c`.

- For each chunk $\mathtt{r}^i$, $0 \leq i < n_r$, the attacker calculates a score for each of the candidates for the chunk by using Equation (2.1), viz.

$$\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i] = n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta),$$

  where the n^i_{ab} values count occurrences of bits across the i-th chunks, $\mathtt{r}^i$, $\mathtt{c}^i$. Therefore, the attacker will produce a list of chunk candidates L_{r^i} for the chunk $\mathtt{r}^i$, whose number of entries depends on the candidates considered for the chunk (and the underlying field), as was showed above.
- For each block $\mathtt{b}^j$, $0 \leq j < n_b$, the attacker will present all the lists of chunk candidates corresponding to all the chunks in $\mathtt{b}^j$ as inputs to OKEA, which was described in Section 3.2.1, to produce a list of the M_j highest scoring candidates for the block, L_{b^j}.

Once `Phase I` finishes, the attacker will get n_b lists and then proceeds as follows.

`Phase II`

- These lists L_{b^j} are given as inputs to an instance of a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block b^j. This instance will generate high scoring candidates for $\mathtt{r}$. For each complete candidate $\mathtt{r}'$, the attacker will extract the elements $a_{1,1}, a_{2,1}, \ldots, a_{n,1}, a_{1,2}, \ldots, a_{n,n}, c_1, \ldots c_n$ from $\mathtt{r}'$[4] to form both a candidate $\mathtt{A}'$ for the matrix $\mathbf{A}$ and a candidate $\mathtt{c}'$ for the vector $\mathbf{c}$. If $\mathtt{A}'$ is invertible, then $(\mathtt{A}', \mathtt{c}')$ is a high scoring candidate for $\mathcal{L}$.

In the `Bouncy Castle` implementation, an affine map $\mathcal{L}$ is represented by two consecutive `short` arrays, namely `Ainv` and `c`. Recall that the attacker gets noisy versions of these two `short` arrays. Let us name them as $\mathtt{Ainv}'$ and $\mathtt{c}'$. The first array has n^2 entries, while the second has n entries. In order to construct high scoring candidates for $\mathcal{L}$, the attacker makes use of the key recovery technique introduced in Section 4.4.1 as follows.

The attacker first regards both $\mathtt{Ainv}'$ and $\mathtt{c}'$ as a single `short` array $\mathtt{r}$. This array $\mathtt{r}$ has n_r entries, where $n_r = n^2 + n$. Besides the attacker regards a `short` entry as a chunk. Therefore, $W = 16 \cdot n_r$ and $w = 16$. Now recall that this implementation uses $\mathbb{F}_{256}$ as a field and that each field element is represented by an 8 bit-string that is stored in the 8

[4]This algorithm depends on the underlying field K.

least significant bits of a `short` variable. Therefore, the candidates for a chunk may be seen as all the integers in the range from 0 to 255.

Furthermore, the attacker represents $\mathtt{r}$ as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j}, with $n_{b^j} > 0$, consecutive chunks, such that $n_r = \sum_{j=0}^{n_b-1} n_{b^j}$. Therefore,

$$\mathtt{r} = \mathtt{b}^0||\mathtt{b}^1||\cdots||\mathtt{b}^{n_b-1},$$

where

$$\mathtt{b}^j = \mathtt{r}^{i_j}||\mathtt{r}^{i_j+1}||\cdots||\mathtt{r}^{i_j+n_{b^j}-1},$$

for $0 \leq j < n_b$ and some $0 \leq i_j < n_r$. The attacker now proceeds as follows.

`Phase I`

- For a chunk $\mathtt{r}^i$, $0 \leq i < n_r$, the attacker will compute a score for each of the 256 candidates for the chunk by using Equation (2.1). Hence, the attacker will obtain a list of chunk candidates L_{r^i} with 256 entries for the chunk $\mathtt{r}^i$.
- For each block $\mathtt{b}^j$, $0 \leq j < n_b$, the attacker presents all the lists of chunk candidates corresponding to all the chunks in $\mathtt{b}^j$ as inputs to OKEA to produce a list of the M_j highest scoring candidates for the block, L_{b^j}.

Once `Phase I` finishes, the attacker will get n_b lists and then proceeds as follows.

`Phase II`

- These lists L_{b^j} are given as inputs to an instance of a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block b^j. This instance will generate high scoring candidates for $\mathtt{r}$. For each complete candidate $\mathtt{r}'$, the attacker will retrieve the first n^2 entries from $\mathtt{r}'$ to form a candidate $\mathtt{Ainv}'$ for the inverse matrix of $\mathbf{A}$ and retrieve the remaining n entries from $\mathtt{r}'$ to form a candidate $\mathtt{c}'$ for the vector $\mathbf{c}$. If $\mathtt{Ainv}'$ is invertible, then $(\mathtt{Ainv}', \mathtt{c}')$ is a high scoring candidate for $\mathcal{L}$.

6.4.1.2 Constructing High Scoring Candidates for f.

Here we will present algorithms to construct a list of high scoring candidates for the central map f. These algorithms may be executed to carry out the three step of the key recovery algorithm introduced in Section 6.4.1.

Let us assume the central map f consists of n_{layer} layers. To construct a list of high scoring candidates for f, the attacker may proceed as follows.

1. For each layer i, $1 \leq i \leq n_{layer}$, the attacker constructs a list of high scoring candidates for the layer i, L_{l_i}.
2. The attacker then presents these lists L_{l_i} as inputs to an instance of an enumeration algorithm, regarding each list L_{l_i} as a set of candidates for the layer i. This instance will pick a candidate from each L_{l_i} and form a candidate for f.

The previous algorithm requires to be adapted to each particular implementation, especially its first step.

In the `Reference` implementation, the central map f consists of two layers, i.e., $n_{layer} = 2$. Recall the layer 1 is stored in three consecutive `byte` arrays, namely $\texttt{l1}_o$, $\texttt{l1}_{vo}$ and $\texttt{l1}_{vv}$, while the layer 2 is stored in three consecutive `byte` arrays, namely $\texttt{l2}_o$, $\texttt{l2}_{vo}$ and $\texttt{l2}_{vv}$. Let us name their noisy versions as $\texttt{l1}'_o$, $\texttt{l1}'_{vo}$, $\texttt{l1}'_{vv}$, $\texttt{l2}'_o$, $\texttt{l2}'_{vo}$ and $\texttt{l2}'_{vv}$. And let us consider the case for the layer 1. In order to construct high scoring candidates for the layer 1, the attacker makes use of the key recovery technique introduced in Section 4.4.1 as follows.

The attacker first regards the noisy versions $\texttt{l1}'_o$, $\texttt{l1}'_{vo}$ and $\texttt{l1}'_{vv}$ as a `byte` array `r`. This array has n_r entries, where n_r depends on the underlying field K. Indeed, let $n_{es} = o_1(\frac{v_1(v_1+1)}{2} + v_1 + 1) + o_1^2 \cdot v_1 + o_1^2$. If $K = \mathbb{F}_{16}$, then $n_r = \lceil n_{es}/2 \rceil$ entries. If $K = \mathbb{F}_{31}$, then $n_r = 2 \cdot (n_{es} \text{ div } 3) + n_{es} \mod 3$. And if $K = \mathbb{F}_{256}$, then $n_r = n_{es}$. The attacker now regards each `byte` entry as a chunk. Therefore, $W = 8 \cdot n_r$ and $w = 8$. So the candidates for a chunk depends on the field we are working on, as showed in Section 6.4.1.1.

Furthermore, the attacker represents `r` as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j}, with $n_{b^j} > 0$, consecutive chunks, such that $n_r =$

$\sum_{j=0}^{n_b-1} n_{b^j}$. Therefore,

$$\mathtt{r} = \mathtt{b}^0||\mathtt{b}^1||\cdots||\mathtt{b}^{n_b-1},$$

where

$$\mathtt{b}^j = \mathtt{r}^{k_j}||\mathtt{r}^{k_j+1}||\cdots||\mathtt{r}^{k_j+n_{b^j}-1},$$

for $0 \leq j < n_b$ and some $0 \leq k_j < n_r$. In order to construct a list of high scoring candidates for this layer, the attacker does the following:

`Phase I`

- For a chunk $\mathtt{r}^k$, $0 \leq k < n_r$, the attacker will compute a score for each of the candidates for the chunk by using Equation (2.1). Hence, the attacker will obtain a list of chunk candidates L_{r^k} for the chunk $\mathtt{r}^k$, the size of this list depends on the number of candidates considered for this chunk (see Section 6.4.1.1).
- For each block $\mathtt{b}^j$, $0 \leq j < n_b$, the attacker presents $L_{r^{k_j}}, \ldots, L_{r^{k_j+n_{b^j}-1}}$ as inputs to an instance of OKEA to produce a list of the M_j highest scoring candidates for the block, L_{b^j}.

Once `Phase I` finishes, the attacker will get n_b lists and then proceeds as follows.

`Phase II`

- The attacker then presents these lists L_{b^j} as inputs to an instance of a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block $\mathbf{b}^j$. This instance will pick a candidate from each L_{b^j} and form a candidate for the array $\mathtt{r}$. For each complete candidate $\mathtt{r}'$, the attacker will extract the 'o-linear' part, the 'vo' part and the 'vv' part to form a candidate for the layer 1.

In the `Bouncy Castle` implementation, the central map f consists of n_{layer} layers, where $n_{layer} \geq 1$. Recall the layer l, $1 \leq l \leq n_{layer}$, is stored in four consecutive `short` arrays, namely $\mathtt{coeff\_alpha}^{(l)}$, $\mathtt{coeff\_beta}^{(l)}$, $\mathtt{coeff\_gamma}^{(l)}$ and $\mathtt{coeff\_eta}^{(l)}$. Let us name their noisy versions as $\mathtt{coeff\_alpha}'^{(l)}$, $\mathtt{coeff\_beta}'^{(l)}$, $\mathtt{coeff\_gamma}'^{(l)}$ and $\mathtt{coeff\_eta}'^{(l)}$. In order to construct high scoring candidates for the layer l, the attacker makes use of the key recovery technique introduced in Section 4.4.1 as follows.

The attacker first regards the noisy versions $\mathtt{coeff\_alpha}'^{(l)}$, $\mathtt{coeff\_Beta}'^{(l)}$, $\mathtt{coeff\_gamma}'^{(l)}$

and $\texttt{coeff\_eta}'^{(l)}$ as a `short` array $\mathtt{r}$ with $n_r = o_l v_l^2 + o_l^2 v_l + o_l(o_l + v_l) + o_l$ entries. Besides, the attacker regards each `short` entry as a chunk. Therefore, $W = 16 \cdot n_r$ and $w = 16$. Now recall that this implementation uses $\mathbb{F}_{256}$ as a field and that each field element is represented by an 8 bit-string that is stored in the 8 least significant bits of a `short` variable. Therefore, the candidates for a chunk may be seen as all the integers in the range from 0 to 255.

Moreover, the attacker represents $\mathtt{r}$ as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j}, with $n_{b^j} > 0$, consecutive chunks, such that $n_r = \sum_{j=0}^{n_b-1} n_{b^j}$. Therefore,

$$\mathtt{r} = \mathtt{b}^0 || \mathtt{b}^1 || \cdots || \mathtt{b}^{n_b-1},$$

where

$$\mathtt{b}^j = \mathtt{r}^{k_j} || \mathtt{r}^{k_j+1} || \cdots || \mathtt{r}^{k_j+n_{b^j}-1},$$

for $0 \leq j < n_b$ and some $0 \leq k_j < n_r$. In order to construct a list of high scoring candidates for the layer l, the attacker does the following.

`Phase I`

- For a chunk $\mathtt{r}^k$, $0 \leq k < n_r$, the attacker will compute a score for each of the 256 candidates for the chunk by using Equation (2.1). Hence, the attacker will obtain a list of chunk candidates L_{r^k} with 256 entries for the chunk $\mathtt{r}^k$.
- For each block $\mathtt{b}^j$, the attacker presents $L_{r^{k_j}}, \ldots, L_{r^{k_j+n_{b^j}-1}}$ as inputs to an instance of OKEA, which was described in Section 3.2.1, to produce a list of the M_j highest scoring candidates for the block $\mathtt{b}^j$, L_{b^j}.

Once `Phase I` finishes, the attacker will get n_b lists and then proceeds as follows.

`Phase II`

- The attacker then presents these lists L_{b^j} as inputs to an instance of a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block $\mathbf{b}^j$. This instance will pick a candidate from each L_{b^j} and form a candidate for $\mathtt{r}$. For each complete candidate $\mathtt{r}^{(l)}$, the first $o_l v_l^2$ entries represent a candidate for the matrix $\texttt{coeff\_alpha}^{(l)}$, the next $o_l^2 v_l$ entries represent a candidate for the

matrix $\texttt{coeff\_beta}^{(l)}$, the next $o_l(v_l + o_l)$ entries represent a candidate for the matrix $\texttt{coeff\_gamma}^{(l)}$ and the remaining o_l entries represent a candidate for the array $\texttt{coeff\_eta}^{(l)}$.

6.4.1.3 Memory Consumption

Here we will describe how much space in memory the key recovery algorithm introduced in Section 6.4.1 will consume during its execution.

The analysis is straight-forward. After the first, second and third steps of this algorithm have been carried out, the algorithm will have stored $|C^{(\mathcal{L}_1)}|$ candidates for $\mathcal{L}_1$, $|C^{(\mathcal{L}_2)}|$ candidates for $\mathcal{L}_2$ and $|C^{(f)}|$ candidates for f in memory. Furthermore, after the fourth step is carried out, the table $\mathcal{S}$ will have $|C^{(\mathcal{L}_1)}|$ entries, each of which is of the form $(\mathbf{y}, i)$, where $\mathbf{y} \in K^{n-v_1}$ and i is an integer. And after the fifth step is carried out, the table $\mathcal{T}$ will have $|C^{(\mathcal{L}_2)}|$ entries, each of which is of the form $(\mathbf{y}', j)$, where $\mathbf{y}' \in K^n$ and j is an integer.

Let $B_{\mathcal{L}_1}$,$B_{\mathcal{L}_2}$ and B_f be the number of bits a candidate for $\mathcal{L}_1$, $\mathcal{L}_2$ and f respectively occupies in memory. Let $B_{\mathcal{S}}$ and $B_{\mathcal{T}}$ be the number of bits an entry in $\mathcal{S}$ and $\mathcal{T}$ respectively occupies in memory. Therefore, the number of bits this algorithm consumes during its execution is

$$|C^{(\mathcal{L}_2)}| \cdot B_{\mathcal{L}_1} + |C^{(\mathcal{L}_1)}| \cdot B_{\mathcal{L}_2} + |C^{(f)}| \cdot B_f + |C^{(\mathcal{L}_1)}| \cdot B_{\mathcal{S}} + |C^{(\mathcal{L}_2)}| \cdot B_{\mathcal{T}}.$$

In the `Reference` implementation, a candidate for $\mathcal{L}_1$ is represented as a sequence of $(n-v_1)^2+(n-v_1)$ field elements, a candidate for $\mathcal{L}_2$ is represented as a sequence of n^2+n field elements. Moreover, a candidate for f consists of both a candidate for layer 1 and a candidate for layer 2. A candidate for layer 1 is represented as a sequence of n_{l_1} field elements, where $n_{l_1} = o_1 + o_1 \cdot v_1 + o_1(\frac{v_1(v_1+1)}{2} + v_1 + 1)$; while a candidate for layer 2 is represented as a sequence of n_{l_2} field elements, where $n_{l_2} = o_2+o_2 \cdot v_2+o_2(\frac{v_2(v_2+1)}{2}+v_2+1)$. Now an entry in $\mathcal{S}$ can be represented as sequence of $n - v_1$ field elements plus an integer (stored in 32 bits). Similarly, an entry in $\mathcal{T}$ can be represented as sequence of n field elements plus an integer. Therefore, If $K = \mathbb{F}_{16}$, then $B_{\mathcal{L}_1} = 8 \cdot \lceil((n-v_1)^2 + (n-v_1))/2\rceil$, $B_{\mathcal{L}_2} = 8 \cdot \lceil(n^2+n)/2\rceil$, $B_f = 8 \cdot \lceil(\sum_{i=1}^{2} n_{l_i})/2\rceil$, $B_{\mathcal{S}} = 8 \cdot \lceil(n-v_1)/2\rceil+32$, $B_{\mathcal{T}} = 8 \cdot \lceil n/2\rceil+32$ Moreover, when $K = \mathbb{F}_{256}$, then $B_{\mathcal{L}_1} = 8 \cdot ((n-v_1)^2 + (n-v_1))$, $B_{\mathcal{L}_2} = 8 \cdot (n^2+n)$,

$B_f = 8 \cdot \sum_{i=1}^{2} n_{l_i}$, $B_{\mathcal{S}} = 8 \cdot (n - v_1) + 32$, $B_{\mathcal{T}} = 8n + 32$. Finally, when $K = \mathbb{F}_{31}$, then

$B_{\mathcal{L}_1} = (((n - v_1)^2 + (n - v_1)) \text{ div } 3) \cdot 16 + (((n - v_1)^2 + (n - v_1)) \mod 3) \cdot 8.$
$B_{\mathcal{L}_2} = ((n^2 + n) \text{ div } 3) \cdot 16 + ((n^2 + n) \mod 3) \cdot 8.$
$B_f = ((\sum_{i=1}^{2} n_{l_i}) \text{ div } 3) \cdot 16 + ((\sum_{i=1}^{2} n_{l_i}) \mod 3) \cdot 8.$
$B_{\mathcal{S}} = ((n - v_1) \text{ div } 3) \cdot 16 + ((n - v_1) \mod 3) \cdot 8 + 32.$
$B_{\mathcal{T}} = (n \text{ div } 3) \cdot 16 + (n \mod 3) \cdot 8 + 32.$

In the `Bouncy Castle` implementation, a candidate for $\mathcal{L}_1$ is represented as a sequence of $(n-v_1)^2 + (n-v_1)$ field elements, while a candidate for $\mathcal{L}_2$ is represented as a sequence of $n^2 + n$ field elements. Moreover, a candidate for f consists of n_{layer} layer candidates. A layer candidate for layer l, $1 \leq l \leq n_{layer}$, is represented as a sequence of n_{l_l} field elements, where $n_{l_l} = o_l v_l^2 + o_l^2 v_l + o_l(v_l + o_l) + o_l$. Now an entry in $\mathcal{S}$ can be represented as sequence of $n - v_1$ field elements plus an integer (stored in 32 bits). Similarly, an entry in $\mathcal{T}$ can be represented as sequence of n field elements plus an integer. This implementation only works using $K = \mathbb{F}_{256}$ and stores a field element in the 8 least significant bits of a `short` variable (there are no packing techniques). Therefore, $B_{\mathcal{L}_1} = 16 \cdot ((n - v_1)^2 + (n - v_1))$, $B_{\mathcal{L}_2} = 16 \cdot (n^2 + n)$, $B_f = 16 \cdot \sum_{i=1}^{n_{layer}} n_{l_l}$, $B_{\mathcal{S}} = 16 \cdot (n - v_1) + 32$, $B_{\mathcal{T}} = 16 \cdot n + 32$.

6.4.1.4 Running Time

Here we will analyse the running time of the key recovery algorithm introduced in Section 6.4.1. Let us assume that the lists $C^{(\mathcal{L}_1)}$, $C^{(\mathcal{L}_2)}$ and $C^{(f)}$ have been produced by the algorithm, i.e., the first three steps of the algorithm have been carried out successfully. Let $T_{AM}(n)$, $T_f(n)$ be the costs for an affine map evaluation and a central map evaluation respectively, with n being the number entries in the input vector.

The fourth step of the algorithm basically proceeds as follows. While iterating through each $h_i \in C^{(\mathcal{L}_1)}$, it computes $\mathbf{y} = h_i^{-1}(H(m))$ and stores $\mathbf{y}$ and i in the table $\mathcal{S}$. Therefore, the cost for this step is given by $|C^{(\mathcal{L}_1)}| \cdot (T_{AM}(n - v_1) + c_1)$, with c_1 being a bound on the number of operations to store an element in table $\mathcal{S}$. Therefore, the cost is $\mathcal{O}(|C^{(\mathcal{L}_1)}|)$. Similarly, the fifth step of the algorithm iterates through each $g_j \in C^{(\mathcal{L}_2)}$, while computing $\mathbf{y}' = g_j(\mathbf{x}')$ and storing $\mathbf{y}'$ and j in the table $\mathcal{T}$. Therefore, the cost for this step is $T_{AM}(n) \cdot (|C^{(\mathcal{L}_2)}| + c_2)$, with c_2 being a bound on the number of operations to store an element in table $\mathcal{T}$. Therefore, the cost is $\mathcal{O}(|C^{(\mathcal{L}_2)}|)$.

The last step of the algorithm iterates through each $f' \in C^{(f)}$ and each $\mathbf{y}' \in \mathcal{T}$, while computing $\mathbf{r} = f'(\mathbf{y}')$ and looking $\mathbf{r}$ up in the table $\mathcal{S}$ to find a valid match. Therefore, the costs of this step is $|C^{(f)}| \cdot |C^{(\mathcal{L}_2)}| \cdot (T_f(n) + c_3)$, with c_3 being a bound on the number of operations to look an element up in the table $\mathcal{S}$. Therefore, the cost of this step is $\mathcal{O}(|C^{(f)}| \cdot |C^{(\mathcal{L}_2)}|)$.

6.4.1.5 Parallelisation

The fourth, fifth and sixth steps of our key recovery algorithm are amenable to parallelisation. Indeed, let us suppose we want to have t independent tasks $T_1, T_2, \ldots, T_t$ executed in parallel.

As for the fourth step, we partition $C^{(\mathcal{L}_1)}$ into t sublists $C_1^{(\mathcal{L}_1)}, \ldots, C_t^{(\mathcal{L}_1)}$ and then set the task T_u to run through each $h_{u_i} \in C_u^{(\mathcal{L}_1)}$ while computing $\mathbf{y_{u_i}} = h_{u_i}^{-1}(H(m))$ and storing the 2-tuple $(\mathbf{y_{u_i}}, u_i)$ in the table $\mathcal{S}$. Likewise this parallelisation strategy may be applied to the fifth step of the recovery algorithm.

Regarding the sixth step of the key recovery algorithm, the strategy is basically the same. We partition $C^{(f)}$ into t sublists $C_1^{(f)}, \ldots, C_t^{(f)}$ and then set the task T_u to run through each $f_{u_k} \in C_u^{(f)}$ and each $\mathbf{y}' \in \mathcal{T}$ while computing $\mathbf{r} = f_{u_k}(\mathbf{y}')$ and looking $\mathbf{r}$ up in the table $\mathcal{S}$. If a match is found, then $\mathbf{r} = f_{u_k}(g_j(\mathbf{x}')) \in \mathcal{S}$, for some j, so $f_{u_k}(g_j(\mathbf{x}')) = h_i^{-1}(H(m))$, for some i and j. Hence, the 3-tuple (h_i, g_j, f_{u_k}) is a candidate for the private key.

6.5 Experimental Evaluation

In this section, we will show some results from running the key recovery algorithm to find a private key for a particular parameter set. In particular, we will mainly focus on its success rate.

6.5.1 Implementation

We implement our key recovery algorithm in Java to take advantage of the implemented algorithms previously and other Rainbow-related algorithms provided by the `Bouncy Castle` implementation. Besides, we consider the format this implementation uses to

store a private key in memory.

6.5.2 Setup

We first pick an ad hoc parameter set to learn more about this algorithm. In particular, we set $v_1 = 4$, $o_1 = 4$, $v_2 = 8$, $o_2 = 4$. For these particular values, the affine map $\mathcal{L}_1$ is represented as a `short` array with size 72, while the affine map $\mathcal{L}_2$ is represented as a `short` array with size 156. The central map has two layers. The first and second layer will be represented as a `short` array with sizes 164 and 436 respectively.

As for producing array candidates with size 72 as candidates for $\mathcal{L}_1$, we set a block to be a sequence of 9 chunks, hence resulting in $n_b = 8 = 72/9$ blocks. Besides, we set the number of candidates to be generated for each block, $\texttt{blsize}_1$, to 256. So for each block $\mathtt{b}^j$, $0 \leq j < n_b$, OKEA will produce a list of the 256 highest scoring candidates L_{b^j} in `Phase I`. Once `Phase I` has completed, the resulting block lists will be given as inputs to an instance of the key enumeration algorithm described in Section 5.6.2.3. Finally, this instance will compute an interval of the form $[max - \delta, max]$ such that the number of candidates $N_{\mathcal{L}_1}$ whose scores lie in the interval is at least a value $\texttt{lsize}_1$ but the difference $N_{\mathcal{L}_1} - \texttt{lsize}_1$ is the smallest.[5] This parameter $\texttt{lsize}_1$ may assume the values $2^{20}, 2^{30}, 2^{40}$. Therefore, once `Phase II` has completed, we have $|C^{(\mathcal{L}_1)}| \leq N_{\mathcal{L}_1}$.

Regarding the creation of array candidates with size 156 as candidates for $\mathcal{L}_2$, we set a block to be a sequence of 39 chunks, hence resulting in $n_b = 4 = 156/39$ blocks. Besides, we set the number of candidates to be generated for each block, $\texttt{blsize}_2$, to 1024. So for each block $\mathtt{b}^j$, $0 \leq j < n_b$, OKEA will produce a list of the 1024 highest scoring candidates L_{b^j} in `Phase I`. Once `Phase I` has completed, the resulting block lists are given as inputs to an instance of the key enumeration algorithm described in Section 5.6.2.3. Finally, this instance will compute an interval of the form $[max - \delta, max]$ such that the number of candidates $N_{\mathcal{L}_2}$ whose scores lie in the interval is at least $\texttt{lsize}_2$ but the difference $N_{\mathcal{L}_2} - \texttt{lsize}_2$ is the smallest, where $\texttt{lsize}_2 \in \{2^{20}, 2^{30}, 2^{40}\}$. Hence, after `Phase II` has completed, we have $|C^{(\mathcal{L}_2)}| \leq N_{\mathcal{L}_2}$.

Concerning the creation of array candidates for f, we regard the first and second layer as a

[5] max is the highest score and is easily calculated by summing the entry 0 of each L_{b^j}. The other value is computed by iterating through the internal list L_{bt} while summing counts and checking when the count is greater than $\texttt{lsize}_1$.

single array of `short` entries whose size is 600. And we set a block to be the concatenation of 100 consecutive chunks, hence resulting in $n_b = 6$ blocks. Moreover, we set the number of candidates to be generated for each block, $\texttt{blsize}_3$, to 2^{12}. So for each block $\texttt{b}^j$, $0 \leq j < n_b$, OKEA will produce a list of the 2^{12} highest scoring candidates L_{b^j} in `Phase I`. Once `Phase I` has completed, the resulting block lists are given as inputs to an instance of the key enumeration algorithm described in Section 5.6.2.3. Finally, this instance will compute an interval of the form $[max - \delta, max]$ such that the number of candidates N_f whose scores lie in the interval is at least $\texttt{lsize}_3$ but the difference $N_f - \texttt{lsize}_3$ is the smallest, where $\texttt{lsize}_3 \in \{2^{20}, 2^{30}, 2^{40}\}$. Hence, after `Phase II` has completed, we have $|C^{(f)}| \leq N_f$.

6.5.3 Success Rate

Regarding the success rate of our key recovery algorithm, we first notice that this algorithm will be able to recover the correct private key if and only if the next three conditions hold:

1. The correct candidate for $\mathcal{L}_1$ is in $C^{(\mathcal{L}_1)}$ after the step 1 has been carried out.
2. The correct candidate for $\mathcal{L}_2$ is in $C^{(\mathcal{L}_2)}$ after the step 2 has been carried out.
3. The correct candidate for f is in $C^{(f)}$ after the step 3 has been carried out.

So the success rate for our key recovery algorithm is given by

$$p_{success} = p_{s_1} \cdot p_{s_2} \cdot p_{s_3},$$

with $p_{s_i}, 1 \leq i \leq 3$, being the probability such that the condition i is satisfied.

We estimate the values $p_{s_1}, p_{s_2}, p_{s_3}$ for our selected parameter set for data points $(\alpha, \beta) \in \{0.001, 0.002, 0.003, 0.004, 0.005\}^2$, $\texttt{lsize}_k \in \{2^{20}, 2^{30}, 2^{40}\}$, $k \in \{1, 2, 3\}$, and the full enumeration in `Phase II`. To achieve such an estimation of these values, we run an auxiliary algorithm 100 times.

For each iteration the auxiliary algorithm first generates a fresh private key and perturbs each component of it according to α, β. The algorithm then runs a tweaked key recovery algorithm $\mathbb{A}$ for $\mathcal{L}_1$ with the corresponding parameters, then runs it for $\mathcal{L}_2$ with the corresponding parameters and finally runs it for f with the corresponding parameters.

In a run of the algorithm $\mathbb{A}$, it first receives the original `short` array `orginal` of a component of the private key,[6] the corresponding noisy version `noisy`, the set $\{2^{20}, 2^{30}, 2^{40}\}$, the corresponding array `blimits`, which contains the indices of the last chunks of each block, and the corresponding number of candidates to be generated for each block, `blsize`. $\mathbb{A}$ then partitions `noisy` into n_b blocks, using `blimits`, viz. $\texttt{noisy}^0||\texttt{noisy}^1||\cdots||\texttt{noisy}^{n_b-1}$, and then run `Phase I` on `noisy`, hence producing the corresponding lists of chunk candidates, $L_{b^j}, 0 \leq j < n_b$, each having `blsize` chunk candidates. The algorithm $\mathbb{A}$ then proceeds to split the original array `orginal` into n_b blocks, using `blimits`, viz. $\texttt{orginal}^0||\texttt{orginal}^1||\cdots||\texttt{orginal}^{n_b-1}$ and checks if the sub-arrays $\texttt{orginal}^j$ are contained in the corresponding lists L_{b^j}, while accumulating their chunk scores. If the check does not succeed, it exits. Or else, $\mathbb{A}$ then counts a success for the full enumeration since if `Phase II` is run completely, the correct array candidate will be output and inserted into the corresponding list. The algorithm $\mathbb{A}$ then continues operating and, for each $\texttt{lsize} \in \{2^{20}, 2^{30}, 2^{40}\}$, it then proceeds to compute an interval $[max - \delta, max]$ (exploiting the key enumeration algorithm described in Section 5.6.2.3) and verify if the computed total score for the original array lies in the computed interval. If so, it means that if `Phase II` is run over the computed interval, the correct array candidate will be output and inserted into the corresponding set (therefore counting a success for the particular value of `lsize`).

Table 6.1 shows the results only for data points (α, β) for which we can get a non-zero $p_{success}$. For example, for $\alpha = 0.001, \beta = 0.001, \texttt{lsize}_1 = 2^{40}, \texttt{lsize}_2 = 2^{40}, \texttt{lsize}_3 = 2^{40}$, then $p_{success} = (0.97) \cdot (0.86) \cdot (0.26) = 0.224458$ and the cost of our key recovery would be at most 2^{80} operations. We may observe that this recovery algorithm manages to tolerate very small values for both α and β. We believe that this behaviour is a result of the inherent design of the Rainbow signature scheme, regardless of its implementation. Indeed, the sizes of the `short` arrays used to store the components of the private key are considerably large, even for this ad hoc parameter set. Even though our recovery algorithm exploits the further redundancy used to store a field element (only 8 bits being used out of 16 bits), both the number of entries per array and the number of candidates per chunk make it difficult for our key recovery algorithm to tolerate larger values for both α, β.

When using the default parameter sets for this scheme, we also expect to obtain very small

[6] Array representation of $\mathcal{L}_1$ or $\mathcal{L}_2$ or f.

Data Points		$\texttt{lsize}_1$				$\texttt{lsize}_2$			$\texttt{lsize}_3$			
α	β	2^{20}	2^{30}	2^{40}	2^{64}	2^{20}	2^{30}	2^{40}	2^{20}	2^{30}	2^{40}	2^{72}
0.001	0.001	0.97	0.97	0.97	0.97	0.89	0.89	0.89	0.1	0.22	0.26	0.34
0.001	0.002	0.94	0.96	0.96	0.96	0.74	0.77	0.77	0.01	0.03	0.06	0.09
0.001	0.003	0.92	0.97	0.97	0.97	0.46	0.52	0.52	0	0.01	0.02	0.05
0.001	0.004	0.79	0.86	0.87	0.87	0.32	0.39	0.39	0	0	0.01	0.01
0.001	0.005	0.77	0.88	0.89	0.89	0.28	0.37	0.39	0	0	0.01	0.01
0.002	0.001	0.97	0.99	0.99	0.99	0.74	0.77	0.79	0.05	0.08	0.1	0.11
0.002	0.002	0.9	0.93	0.94	0.94	0.54	0.57	0.57	0.01	0.01	0.01	0.03
0.002	0.003	0.82	0.9	0.9	0.9	0.41	0.45	0.46	0	0	0.01	0.01
0.003	0.001	0.89	0.92	0.92	0.92	0.52	0.58	0.59	0.01	0.04	0.07	0.07

Table 6.1: Values for $p_{s_i}, 1 \leq i \leq 3$, for the ad hoc parameter set.

values for $p_{s_1}, p_{s_2}, p_{s_3}$, even if the values for both α and β are very small. Recall that in the `Bouncy Castle` implementation, the default parameter set is the following.

$$v_1 = 6 \; ; o_1 = 12 - 6 = 6,$$
$$v_2 = 12; o_2 = 17 - 12 = 5,$$
$$v_3 = 17; o_3 = 22 - 17 = 5,$$
$$v_4 = 22; o_4 = 33 - 22 = 11,$$
$$v_5 = 33; o_5 = 0.$$

When using this parameter set, our key recovery algorithm will first attempt to construct a set of high scoring candidates for L_1, $C^{(\mathcal{L}_1)}$. The affine map $\mathcal{L}_1$ will be represented as a `short` array with size $(n - v_1) \cdot (n - v_1 + 1) = (33 - 6)(33 - 6 + 1) = 27 \cdot 28 = 756$. The construction of such a set involves generating high scoring candidates for an array `r` with size equal to 756. Hence, p_{s_1} is expected to be very small, even if the values for both α and β are very small. Similarly, the second step of our key recovery algorithm involves constructing a set of high scoring candidates for L_2, $C^{(\mathcal{L}_2)}$. The affine map L_2 will be represented as a `short` array with size $n^2 + n = (33)(33) + 33 = 1122$. Hence, p_{s_2} is also expected to be very small, even if the values for both α and β are very small. Concerning

the third step, the probability for this step to complete successfully will be even smaller, because the attacker would need to create a set of high scoring candidates for f such that the correct candidate is in the corresponding list $C^{(f)}$. A high scoring candidate for f results from picking a high scoring candidate for each layer and then combining them. In particular, the first, second, third and fourth layer will be represented as a `short` array with sizes 510, 1110 1985 and 8360 respectively. Therefore, p_{s_3} is expected to be very small, even if the values for both α and β are very small. In conclusion, the probability $p_{success}$ for our key recovery algorithm to recover the correct private key is expected to be very small for the default parameter set in `Bouncy Castle`.

On the hand, let us consider the default parameter set $(\mathbb{F}_{16}, 32, 32, 32)$ introduced in the `Reference` implementation and the particular encoding of this implementation. The reason for choosing this parameter set is that it represents one of the "smallest" parameter sets. So, according to this parameter set, $v_1 = 32$, $o_1 = 32$, $v_2 = 64$, $o_2 = 32$, $n = 96$. In this implementation, the affine map $\mathcal{L}_1$ is represented as a `byte` array whose size is equal to $\lceil (n-v_1)\cdot(n-v_1+1)/2 \rceil = 2080$. Moreover, the affine map $\mathcal{L}_2$ is represented as a `byte` array whose size is equal to $\lceil (n)\cdot(n+1)/2 \rceil = 4656$. The first layer is represented as a `byte` array whose size is equal to $\lceil (o_1 + o_1 \cdot v_1 + o_1(\frac{v_1(v_1+1)}{2} + v_1 + 1))/2 \rceil = 9504$, while the second layer is represented as a `byte` array whose size is equal to $\lceil (o_1 + o_1 \cdot v_2 + o_2(\frac{v_2(v_2+1)}{2} + v_2 + 1))/2 \rceil = 35360$. Because both the number of entries per array and the number of candidates per chunk are considerably large, and this encoding decreases redundancy used to store the private key, we also expect that the probability $p_{success}$ for our key recovery algorithm to recover the correct private key be very small, even if the values for both α and β are very small.

6.6 Chapter Conclusions

In this chapter, we initiated the study of cold boot attacks on the Rainbow signature scheme, a member of the family of asymmetric cryptographic primitives based on multivariate polynomials over a finite field K. Our evaluation focused on two existing Rainbow implementations. The first is the `Reference` implementation included in the package submitted to the NIST process. The second is the Java implementation included in the popular `Bouncy Castle` Java crypto library. By exploiting the algebraic structure of this scheme, we proposed a time-memory trade-off algorithm for key recovery and then exper-

imented with this algorithm for an ad hoc parameter set. We found that our algorithm is not able to tolerate larger values for both α, β. We believe this is a result of the inherent design of the Rainbow signature scheme, regardless of its implementation. Indeed, the sizes of the arrays used to store the components of the private key are considerably large, even for the ad hoc parameter set we used in our experiments. Therefore, as a future work, it would be interesting to explore other approaches exploiting better the algebraic structure of this scheme.

Chapter 7

Cold Boot Attacks on McEliece Public Key Encryption Scheme

Contents

7.1 Introduction 159
7.2 Goppa Codes 161
7.3 McEliece Public Key Encryption Scheme 169
7.4 Software Implementations for McEliece Public Key Encryption Scheme 170
7.5 Mounting Cold Boot Attacks 175
7.6 Experimental Evaluation 182
7.7 Chapter Conclusions 187

In this chapter, we study the McEliece public key encryption scheme in the cold boot attack setting. In particular, we focus on a implementation provided by the `Bouncy Castle` *project. For this implementation, we study its in-memory formats for storing a private key for this scheme and propose a key recovery strategy that combines key enumeration algorithms with some linear algebra algorithms.*

7.1 Introduction

In this chapter, we examine the feasibility of cold boot attacks against the McEliece public key encryption scheme [51]. This scheme is a member of the family of asymmetric cryptographic primitives based on linear codes. We believe this to be the first time that this has been attempted, however there are very recent studies attempting other side-channel attacks on this scheme [1, 63]. Our work is the continuation of the trend to

develop cold boot attacks for different schemes as revealed by the literature discussed at length in Section 2.2. But it is also the continuation of the evaluation of the leading post-quantum candidates against this class of attack. Such an evaluation should form a small but important part of the overall assessment of schemes in the NIST selection process for post-quantum algorithms.[1]

As noted previously, in the cold boot attack setting, it is important to have knowledge of the exact formats in which the private key of a cryptographic scheme is stored in memory to developing key recovery attacks for the cryptographic scheme. This is because the attack depends on physical effects in memory, represented by bit flips in private key bits, and the main input to the attack is a bit-flipped version of the private key. Ergo, it is necessary either to propose natural ways in which a private key would be stored in memory or to review specific implementations of the McEliece public key encryption scheme to learn about what formats are used to store the private key.

As in previous chapters, we adopt reviewing specific implementations of the McEliece public key encryption scheme, and so our evaluation will be focused on the McEliece implementation included in the popular `Bouncy Castle` Java crypto library. In this chapter, we will study this implementation and evaluate it in the cold boot attack setting. In particular, this implementation stores Goppa polynomials directly in memory in coefficient form. Additionally, it stores some components of the private key with some redundancy and additional structure that we can exploit, e.g. the permutations on $\{0, \ldots, n-1\}$ are stored as an integer array containing each number l, $0 \leq l < n$, once and only once.

We propose a key recovery algorithm that exploits the structure of this cryptographic scheme, which allows us to reconstruct the private key of this scheme from some of its components. To reconstruct the components, we make use of some techniques discussed previously. In particular, we use the general key recovery strategy developed in Chapter 4 as a core algorithm to create lists of high scoring candidates for some components of the private key. Once these lists are created, we exploit some algebraic relations among the components to try to reconstruct the private key.

[1] See `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` for details of the NIST process.

7.2 Goppa Codes

In this section, we will present some concepts on Goppa codes.

Let q be a prime and let m and t be positive integers. Let $g(z)$ be a polynomial of degree t over the extension field $\mathbb{F}_{q^m}$, $g(z) = g_0 + g_1 z + \ldots + g_t z^t = \sum_{i=0}^{t} g_i z^i$, and let $\mathcal{L}$ be a subset of $\mathbb{F}_{q^m}$, $\mathcal{L} = \{\gamma_1, \ldots, \gamma_n\} \subseteq \mathbb{F}_{q^m}$.

A Goppa Code $\Gamma(\mathcal{L}, g(z))$ is defined by the Goppa polynomial $g(z)$ and the subset $\mathcal{L}$ such that $g(\gamma_i) \neq 0$ for all $\gamma_i \in \mathcal{L}$. With a vector $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ over $\mathbb{F}_q$ we associate the function

$$R_{\mathbf{c}}(z) = \sum_{i=1}^{n} \frac{c_i}{z - \gamma_i}, \tag{7.1}$$

where $\frac{1}{z-\gamma_i}$ is the unique polynomial such that $(z - \gamma_i)\frac{1}{z-\gamma_i} \equiv 1 \mod g(z)$.

Definition 7.2.1 *The Goppa code $\Gamma(\mathcal{L}, g(z))$ consists of all vectors $\mathbf{c}$ such that*

$$R_{\mathbf{c}}(z) \equiv 0 \mod g(z). \tag{7.2}$$

7.2.1 Parameters of a Goppa Code

The parameters of a code are its size n, its dimension k and its minimum distance d. We will use the notation $[n, k, d]$ Goppa code for a Goppa code with parameters n, k and d. The first parameter, n, is the length of the codewords $\mathbf{c}$ and therefore is fixed by $\mathcal{L}$. For the other two parameters, lower bounds may be derived.

Theorem 7.2.1 *The Goppa code $\Gamma(\mathcal{L}, g(z))$ of size n is a linear code over $\mathbb{F}_q$ with the properties:*

- *The dimension of the code satisfies $k \geq n - mt$.*
- *The minimum distance of the code satisfies $d \geq t + 1$.*

7.2.1.1 Parameters for a special case

We want the minimum distance of the code d to be as large as possible, so that the code can correct r errors if $2r + 1 \leq d$ [38]. There is a special case where the lower bound on d can be raised. This is the case where $\Gamma(\mathcal{L}, g(z))$ is a binary Goppa code such that the polynomial $g(z)$ over $\mathbb{F}_{2^m}$ of degree t has no roots of multiplicity larger than one. A polynomial satisfying this property is called separable, a name that we will use from now on.

Lemma 7.2.2 *Let $\Gamma(\mathcal{L}, g(z))$ be a binary Goppa code with a separable polynomial $g(z)$ of degree t. Then $\Gamma(\mathcal{L}, g(z)) = \Gamma(\mathcal{L}, g^2(z))$.*

Theorem 7.2.3 *Let $\Gamma(\mathcal{L}, g(z))$ be a binary Goppa code with a separable polynomial $g(z)$ of degree t. Then $\Gamma(\mathcal{L}, g(z))$ has a minimum distance d of at least $2t + 1$.*

7.2.2 Parity Check Matrix of a Goppa Code

We need a parity check matrix $\mathbf{H}$ of the code in order to correct errors. Note that $\frac{1}{z-\gamma_i}$ can be seen as a polynomial $p_i(z) \mod g(z)$, viz.

$$\frac{1}{z - \gamma_i} \equiv p_i(z) = p_{i1} + p_{i2}z + \ldots + p_{it}z^{t-1} \mod g(z).$$

We can therefore rewrite Equation (7.2) as $\sum_{i=1}^{n} c_i p_i(z) \equiv 0 \mod g(z)$, hence

$$\sum_{i=1}^{n} c_i p_{ij} = 0, \text{ for } 1 \leq j \leq t.$$

Furthermore, the parity check matrix $\mathbf{H}$ satisfies $\mathbf{c} \cdot \mathbf{H}^t = 0$, therefore

$$\mathbf{H} = \begin{pmatrix} p_{11} & p_{21} & p_{31} & \cdots & p_{n1} \\ p_{12} & p_{22} & p_{32} & \cdots & p_{n2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{1t} & p_{2t} & p_{3t} & \cdots & p_{nt} \end{pmatrix}. \tag{7.3}$$

To determine the factors p_{ij}, we rewrite $p_i(z) \equiv (z-\gamma_i)^{-1} \equiv -\frac{g(z)-g(\gamma_i)}{z-\gamma_i} \cdot g(\gamma_i)^{-1} \mod g(z)$, since $-(z-\gamma_i) \cdot \frac{g(z)-g(\gamma_i)}{z-\gamma_i} \cdot g(\gamma_i)^{-1} \equiv 1 \mod g(z)$.

We now define $h_i := g(\gamma_i)^{-1}$ and recall that $g(z) = g_0 + g_1 z + \ldots + g_t z^t$. Substituting this into the previous equation, we find

$$p_i(z) = -\frac{g_t \cdot (z^t - \gamma_i^t) + \ldots + g_1(z-\gamma_i)}{z-\gamma_i} \cdot h_i. \tag{7.4}$$

The fraction in Equation (7.4) can be rewritten as

$$g_t(z^{t-1} + z^{t-2}\gamma_i + \ldots + \gamma_i^{t-1}) + g_{t-1}(z^{t-2} + z^{t-3}\gamma_+ \ldots + \gamma_i^{t-2}) + \ldots + g_2(z+\gamma_i) + g_1.$$

If we now substitute $p_i(z) = p_{i1} + p_{i1}z + \ldots + p_{it}z^{t-1}$, we find the following expression for each p_{ij},

$$\begin{cases} p_{i1} = -(g_t\gamma_i^{t-1} + g_{t-1}\gamma_i^{t-2} + \ldots g_2\gamma_i + g_1)h_i. \\ p_{i2} = -(g_t\gamma_i^{t-2} + g_{t-1}\gamma_i^{t-3} + \ldots + g_2)h_i. \\ \quad \vdots \quad \vdots \quad \vdots \\ p_{i(t-1)} = -(g_t\gamma_i + g_{t-1}) \cdot h_i. \\ p_{it} = -g_t \cdot h_i. \end{cases}$$

Hence we find that $\mathbf{H} = \mathbf{C} \cdot \mathbf{X} \cdot \mathbf{Y}$, where

$$\mathbf{C} = \begin{pmatrix} -g_t & -g_{t-1} & -g_{t-2} & \ldots & -g_1 \\ 0 & -g_t & -g_{t-1} & \ldots & -g_2 \\ 0 & 0 & -g_t & \ldots & -g_3, \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & -g_t \end{pmatrix}, \tag{7.5}$$

$$\mathbf{X} = \begin{pmatrix} \gamma_1^{t-1} & \gamma_2^{t-1} & \gamma_3^{t-1} & \dots & \gamma_n^{t-1} \\ \gamma_1^{t-2} & \gamma_2^{t-2} & \gamma_3^{t-2} & \dots & \gamma_n^{t-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \gamma_1 & \gamma_2 & \gamma_3 & \dots & \gamma_n \\ 1 & 1 & 1 & \dots & 1 \end{pmatrix}, \mathbf{Y} = \begin{pmatrix} h_1 & 0 & 0 & \dots & 0 \\ 0 & h_2 & 0 & \dots & 0 \\ 0 & 0 & h_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & h_n \end{pmatrix}.$$

7.2.3 Generator Matrix of a Goppa Code

The parity check matrix $\mathbf{H}$ is used for correcting errors, but besides that, we also need a generator matrix to encode and decode messages. A codeword is formed by multiplying a message $\mathbf{m} = (m_1, \dots, m_k)$ with $\mathbf{G}$. Furthermore, the resulting codeword can be corrected using the property $\mathbf{c} \cdot \mathbf{H}^t = 0$ for all $\mathbf{c} \in \Gamma(\mathcal{L}, g(z))$. Therefore, $\mathbf{G} \cdot \mathbf{H}^t = 0$ can be used to find $\mathbf{G}$ from $\mathbf{H}$.

Moreover, to encode a long message, we first write it in blocks of k symbols $\mathbf{m}$ and then multiply $\mathbf{m}$ by the generator matrix $\mathbf{G}$, viz. $(m_1, m_2, \dots, m_k) \cdot \mathbf{G} = (c_1, c_2, c_3, \dots, c_n)$.

7.2.4 Correcting Errors

Let $\mathbf{y}$ be a received word, containing r errors, with $2r + 1 \leq d$ or $r \leq \lfloor \frac{t}{2} \rfloor$ when $d = t + 1$. Therefore, $(y_1, y_2, \dots, y_n) = (c_1, c_2, \dots, c_n) + (e_1, e_2, \dots, e_n)$ with $e_i \neq 0$ in exactly r positions. In order to correct the word and find the codeword $\mathbf{c}$, we have to find the error vector $\mathbf{e}$ and therefore discover the set of error locations $B = \{i \mid e_i \neq 0\}$ and the corresponding values e_i for $i \in B$.

Let us define the error locator polynomial $\sigma(z)$ as

$$\sigma(z) := \prod_{i \in B} (z - \gamma_i), \tag{7.6}$$

and the error evaluator polynomial $\omega(z)$ as

$$\omega(z) := \sum_{i \in B} e_i \prod_{j \in B, j \neq i} (z - \gamma_j). \tag{7.7}$$

From Equation (7.6), it is clear that the set of error locations follows directly from the roots of $\sigma(z)$, since $B = \{i \mid \gamma_i \text{ is a root of } \sigma(z)\}$. Moreover, the syndrome $s(z)$ of the received word $\mathbf{y}$ is defined as

$$\begin{aligned} s(z) &:= \sum_{i=1}^{n} \frac{y_i}{z-\gamma_i} = \sum_{i=1}^{n} \frac{c_i+e_i}{z-\gamma_i} = \sum_{i=1}^{n} \frac{c_i}{z-\gamma_i} + \sum_{i \in B} \frac{e_i}{z-\gamma_i} \\ &\equiv \sum_{i \in B} \frac{e_i}{z-\gamma_i} \mod g(z). \end{aligned} \tag{7.8}$$

Theorem 7.2.4 *Let* $\mathbf{e}$ *be the error vector of weight* r *such that* $r \leq \lfloor \frac{t}{2} \rfloor$. *Let* $\sigma(z), \omega(z), s(z)$ *be as described above. Therefore,*

1. $deg(\sigma(z)) = r$.
2. $deg(\omega(z)) \leq r - 1$.
3. $gcd(\sigma(z), \omega(z)) = 1$.
4. $e_k = \frac{\omega(\gamma_k)}{\sigma'(\gamma_k)}, k \in B$, *where* $\sigma'(z) := \sum_{i \in B} \prod_{j \in B, j \neq i} (z - \gamma_j)$.
5. $\sigma(z)s(z) \equiv \omega(z) \mod g(z)$.

In order to correct errors in a codeword, we have to solve the following equation,

$$\sigma(z)s(z) \equiv \omega(z) \mod g(z). \tag{7.9}$$

Because $g(z)$ is known and the syndrome $s(z)$ can be computed, we have to solve a system of t equations with $2r$ unknowns, namely, $\sigma_0, \sigma_1, \ldots, \sigma_{r-1}$ and $\omega_0, \omega_1, \ldots, \omega_{r-1}$, where $\sigma(z) = \sigma_0 + \sigma_1 z, \ldots, \sigma_{r-1} z^{r-1} + z^r$ and $\omega(z) = \omega_0 + \omega_1 z + \ldots + \omega_{r-1} z^{r-1}$. Since $2r \leq t$, there is a unique solution. Algorithm 23 describes the correction of errors in a Goppa Code.

Algorithm 23 corrects $r \leq \lfloor \frac{t}{2} \rfloor$ errors in a Goppa Code

1: Let $\mathbf{y} = (y_1, \ldots, y_n)$ be a received codeword containing r errors for $2r \leq t$;
2: Compute the syndrome $s(z) = \sum_{i=1}^{n} \frac{y_i}{z-\gamma_i}$;
3: Solve the key equation $\sigma(z)s(z) \equiv \omega(z) \mod g(z)$ by writing

$$\sigma(z) = \sigma_0 + \sigma_1 z, \ldots, \sigma_{r-1} z^{r-1} + z^r;$$

$$\omega(z) = \omega_0 + \omega_1 z + \ldots + \omega_{r-1} z^{r-1};$$

and solving the system of t equations and $2r$ unknowns. If the code is binary, one can take $\omega(z) = \sigma'(z)$;
4: Determine the set of error locations $B = \{i \mid \sigma(\gamma_i) = 0\}$;
5: Compute the error values $e'_i = \frac{\omega(\gamma_i)}{\sigma'(\gamma_i)}$;
6: The error vector $\mathbf{e} = (e_1, \ldots, e_n)$ is formed by setting $e_i = e'_i$ for $i \in B$ and zeros at the remaining positions;
7: The codeword sent is $\mathbf{c} = \mathbf{y} - \mathbf{e}$;

7.2.4.1 Correcting Errors in a Special case

Let $\Gamma(\mathcal{L}, g(z))$ be a binary Goppa code such that the polynomial $g(z)$ over $\mathbb{F}_{2^m}$ is separable and whose degree is t. Hence, as stated by Theorem 7.2.3, $\Gamma(\mathcal{L}, g(z))$ has a minimum distance d of at least $2t + 1$. Therefore, it can correct a maximum of t errors, so we can make changes to the previous ideas to design an algorithm for correcting t errors in this special case. A general and straightforward approach is as follows.

According to Lemma 7.2.2, $\Gamma(\mathcal{L}, g(z)) = \Gamma(\mathcal{L}, g^2(z))$ and, since $g^2(z)$ has degree $2t$, we can then use Algorithm 23 to correct $\lfloor \frac{2t}{2} \rfloor = t$ errors. But, to do so, we first need to compute the parity check matrix $\hat{\mathbf{H}}$ of $\Gamma(\mathcal{L}, g^2(z))$ to compute the syndrome of the received vector. However, there is another known algorithm if $gcd(s(z), g(z)) = 1$. Note this condition is not satisfied in general for $g(z)$ a separable polynomial. However, if we assume $g(z)$ is irreducible, it is always true. We next show how to get an algorithm for correcting errors when $g(z)$ is irreducible.

Since $\Gamma(\mathcal{L}, g(z))$ is a binary code, Equation (7.9) can be transformed to

$$\sigma(z)s(z) \equiv \sigma'(z) \mod g(z). \tag{7.10}$$

Furthermore, we know that $\sigma(z)$ is a polynomial of degree t and therefore can be rewritten as $\sigma(z) = \sigma_0 + \sigma_1 z + \ldots + \sigma_t z^t$. Grouping terms, we have $\sigma(z) = a^2(z) + b^2(z)z$,

where $deg(a(z)) \leq \frac{t}{2}$ and $deg(b(z)) \leq \frac{t-1}{2}$. Moreover, $\sigma'(z)$ can be rewritten as $\sigma'(z) = 2a(z)a'(z) + 2b(z)b'(z)z + b^2(z) = b^2(z)$. Therefore, using Equation (7.10), we find

$$(a^2(z) + b^2(z)z)s(z) \equiv b^2(z) \mod g(z). \tag{7.11}$$

Since $g(z)$ is irreducible, then $gcd(s(z), g(z)) = 1$. Hence, by using the extended euclidean algorithm(see [66]), we can compute a polynomial $h(z)$ such that

$$h(z)s(z) \equiv 1 \mod g(z). \tag{7.12}$$

Combining Equation (7.11) and Equation (7.12), it follows that

$$b^2(z)(z + h(z)) \equiv a^2(z) \mod g(z) \tag{7.13}$$

If $h(z) = z$, then, from Equation (7.12), $s(z)z \equiv 1 \mod g(z)$, so $\sigma(z) = z$ is the solution. Otherwise, there exists a unique polynomial $d(z) \mod g(z)$ such that $d^2(z) = z + h(z)$. Now from Equation (7.13), we know that $b^2(z)d^2(z) \equiv a^2(z) \mod g(z)$, which is equivalent to $b(z)d(z) \equiv a(z) \mod g(z)$, since it is the binary case. This leads to finding $\sigma(z) = a^2(z) + b^2(z)z$ of degree $\leq t$ and thereby correcting t errors. Algorithm 24 describes the correction of errors in a Binary Goppa Code for $g(z)$ irreducible.

Algorithm 24 corrects $r \leq t$ errors for $g(z)$ irreducible over $\mathbb{F}_{2^m}$

Let $\mathbf{y} = (y_1, \ldots, y_n)$ be a received codeword containing r errors for $r \leq t$;
Compute the syndrome $s(z) = \sum_{i=1}^{n} \frac{y_i}{z-\gamma_i}$;
Use the Extended Euclidean Algorithm (see [66]) to find $h(z)$ such that $s(z)h(z) \equiv 1 \mod g(z)$;
if $h(z) = z$ **then**
    $\sigma(z) \leftarrow z$;
else
    Calculate $d(z)$ such that $d^2(z) \equiv h(z) + z \mod g(z)$;
    Find $a(z)$ and $b(z)$ with $b(z)$ of least degree such that $b(z)d(z) \equiv a(z) \mod g(z)$;
    $\sigma(z) \leftarrow a^2(z) + b^2(z)z$;
end if
Determine the set of error locations $B = \{i \mid \sigma(\gamma_i) = 0\}$;
The error vector $\mathbf{e} = (e_1, \ldots, e_n)$ is formed by setting $e_i = 1$ for $i \in B$ and zeros at the remaining positions;
The codeword sent is $\mathbf{c} = \mathbf{y} - \mathbf{e}$;

7.2.5 Decoding a Message

After correcting possible errors in a codeword, we can find the original message by using this expression $(m_1, m_2, \ldots, m_k) \cdot \mathbf{G} = (c_1, c_2, c_3, \ldots, c_n)$, which is equivalent to

$$\mathbf{G}^t \cdot \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_k \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}.$$

Therefore we can find the message vector by reducing

$$(\mathbf{G}^t \mid \mathbf{c}^t) = \left(\begin{array}{c|c} & c_1 \\ & c_2 \\ \mathbf{G}^t & c_3 \\ & \vdots \\ & c_n \end{array}\right) \sim \cdots \sim \left(\begin{array}{c|c} & m_1 \\ & m_2 \\ \mathbf{I}_k & \vdots \\ & m_k \\ \hline & \\ \multicolumn{2}{c}{\mathbf{M}} \end{array}\right), \tag{7.14}$$

where $\mathbf{I}_k$ is the $k \times k$ identity matrix and $\mathbf{M}$ is an $(n-k) \times (k+1)$ matrix.

7.3 McEliece Public Key Encryption Scheme

In this section, we will present the McEliece public key encryption scheme, the first code-based public-key encryption scheme that was introduced by McEliece [51] in 1978. The public key specifies a random binary Goppa code. A cipher-text is a codeword plus random errors, and the private key allows efficient decoding, i.e., extracting the codeword from the cipher-text, as well as identifying and removing the errors.

The security level of McEliece public key encryption scheme has remained remarkably stable, despite dozens of attack papers over 40 years [8]. The original McEliece parameters were designed for only 2^{64} security [51], but this encryption scheme may easily scale up to provide ample security margin against advances in computer technology, including quantum computers. This encryption scheme has prompted a tremendous amount of followup work [10, 54, 65]. For instance, a variant of this encryption scheme is Niederreiter public key encryption scheme [53]. Additionally, McEliece public key encryption scheme is a candidate in the ongoing NIST post-quantum standardisation process.[2]

On the other hand, research on practical questions regarding efficiency and physical security of this encryption scheme has just recently started, in particular research on timing-related side-channel attacks [1, 63]. We here instead follow other direction. We will evaluate real implementations of this encryption scheme in the cold boot attack setting. This question has not yet been addressed so far. To this end, we will evaluate a current implementation of this encryption scheme in this setting and propose a key recovery algorithm for that particular implementation.

7.3.1 Public Key Encryption Scheme

In order to set up an encryption scheme depending on Goppa codes, we could use the following approach, commonly called textbook McEliece cryptosystem.

7.3.1.1 Key Generation

First select an arbitrary separable polynomial $g(z)$ of degree t over $\mathbb{F}_{2^m}$. The Goppa code defined by $\mathcal{L}$ and $g(z)$ has parameters $[n, \geq n-mt, \geq 2t+1]$. Compute the $k \times n$ generator

[2] See `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` for details of the NIST process.

matrix $\mathbf{G}'$ of the Goppa code. Next, choose a random, dense $k \times k$ non-singular matrix $\mathbf{S}$ and a random $n \times n$ permutation matrix $\mathbf{P}$ and compute $\mathbf{G} = \mathbf{S} \cdot \mathbf{G}' \cdot \mathbf{P}$. The public key is the 2-tuple $(\mathbf{G}, t)$, while the private key is the 4-tuple $(g(z), \mathbf{G}', \mathbf{S}, \mathbf{P})$.

7.3.1.2 Encryption

The plaintext space of the scheme is the set of binary strings of length k. To encrypt such a string $\mathbf{m}$, pick a random error pattern $\mathbf{e}$ of length n with at most t coordinates equal to 1. The encryption of $\mathbf{m}$ is then $\mathbf{c} = \mathbf{m} \cdot \mathbf{G} + \mathbf{e}$.

7.3.1.3 Decryption

In order to recover $\mathbf{m}$ from $\mathbf{c}$, we compute

$$\mathbf{c}' = \mathbf{c} \cdot \mathbf{P}^{-1} = \mathbf{m} \cdot \mathbf{G} \cdot \mathbf{P}^{-1} + \mathbf{e} \cdot \mathbf{P}^{-1} = \mathbf{mSG}' \cdot \mathbf{P} \cdot \mathbf{P}^{-1} + \mathbf{e}' = (\mathbf{m} \cdot \mathbf{S}) \cdot \mathbf{G}' + \mathbf{e}',$$

where $\mathbf{e}'$ still has weight at most t, since $\mathbf{P}$ is a permutation. With the decoding algorithm for Goppa codes, we then can correct $\mathbf{c}'$ into the codeword $\mathbf{m}' = \mathbf{m} \cdot \mathbf{S}$ by finding $\mathbf{e}'$ and finally recover the original message by computing $\mathbf{m} = \mathbf{m}' \cdot \mathbf{S}^{-1}$.

7.4 Software Implementations for McEliece Public Key Encryption Scheme

In this section, we will review the software implementation for McEliece public key encryption scheme provided by `Bouncy Castle` crypto package. In particular, we are interested in the in-memory format this implementation uses to store the components of the private key.

7.4.1 `Bouncy Castle` Implementation

The `Bouncy Castle` crypto package contains Java implementations of cryptographic algorithms. The package is organised so that it contains a light-weight API suitable for use in any environment (including the Java 2 platform, Micro Edition) with the additional infrastructure to conform the algorithms to the Java Crytpgraphy Extension framework.

It offers the BCPQ (for BC Post Quantum) provider with support for McEliece public key encryption scheme. We now will describe the in-memory format this implementation uses to store the components of the private key.

The package `org.bouncycastle.pqc.crypto.mceliece` contains all the relevant classes for this implementation. Here we will only describe those classes in the package that are relevant to the key generation algorithm.

We start off by describing the class `McElieceParameters` that defines the default parameters to set up a new key pair. Specifically, it contains the following `int` variables.

1. `DEFAULT_M` holds the default extension degree. It is set to the value 11.
2. `DEFAULT_T` holds the default error correcting capability. It is set to the value 50.
3. `m` stores the extension degree of the finite field $\mathbb{F}_{2^m}$.
4. `t` stores the error correction capability of the code.
5. `n` stores length of the code.
6. `fieldPoly` stores the field polynomial.

During the initialisation of an object of this class, `m` and `t` are set to the default values if they are not specified. Furthermore, `n` is set to 2^m while `fieldPoly` is set to an integer value that represents an irreducible polynomial of degree m over $\mathbb{F}_2$. For example, the irreducible polynomial z^4+z+1 is represented by the integer value $(00...0000010011) = 19$. When the default values are selected, we have $n = 2048$, $k = 1498$ and $t = 50$. This instance of the McEliece cryptosystem gives about 100 bits of security with respect to the attacks given in [8].

Now the class `McElieceKeyPairGenerator` internally maintains an object of the class `McElieceParameters`. Also, it possesses the method `genKeyPair`, which will generate both the private key and the public key when it is executed. Figure 7.1 shows the actual implementation of this method. As we shall see, this key generation algorithm is slightly different to the algorithm described in Section 7.3.1.1.

We next explain the code shown in Figure 7.1 by expanding on the lines of code below each numbered comment.

```
private AsymmetricCipherKeyPair genKeyPair()
{   if (!initialized)
    {initializeDefault();}
    //1. Creating the finite field GF(2^m).
    GF2mField field = new GF2mField(m, fieldPoly);
    //2. Creating an Irreducible Goppa polynomial.
    PolynomialGF2mSmallM gp = new PolynomialGF2mSmallM(field, t,
        PolynomialGF2mSmallM.RANDOM_IRREDUCIBLE_POLYNOMIAL, random);
    //3. Generating the canonical check matrix.
    GF2Matrix h = GoppaCode.createCanonicalCheckMatrix(field, gp);
    //4. Computing both the short systematic form of the check matrix and a
        random permutation P1.
    MaMaPe mmp = GoppaCode.computeSystematicForm(h, random);
    GF2Matrix shortH = mmp.getSecondMatrix();
    Permutation p1 = mmp.getPermutation();
    //5. Computing the short systematic form of the generator matrix and
        extending it to full systematic form.
    GF2Matrix shortG = (GF2Matrix)shortH.computeTranspose();
    GF2Matrix gPrime = shortG.extendLeftCompactForm();
    //6. Obtaining the number of rows of G, i.e. the dimension of the code.
    int k = shortG.getNumRows();
    //7. Generating a random invertible (k x k)-matrix S and its inverse.
    GF2Matrix[] matrixSandInverse = GF2Matrix
        .createRandomRegularMatrixAndItsInverse(k, random);
    //8. Generating a random permutation P2.
    Permutation p2 = new Permutation(n, random);
    //9. Computing the public matrix G=S*G'*P2
    GF2Matrix g = (GF2Matrix)matrixSandInverse[0].rightMultiply(gPrime);
    g = (GF2Matrix)g.rightMultiply(p2);
    //10. Generating both the public key and the private key.
    McElieceCCA2PublicKeyParameters pubKey = new McElieceCCA2PublicKeyParameters(n,
        t, g);
    McElieceCCA2PrivateKeyParameters privKey = new
        McElieceCCA2PrivateKeyParameters(n, k, field, gp, p1, p2,
        matrixSandInverse[1]);
    return new AsymmetricCipherKeyPair(pubKey, privKey);
}
```

Figure 7.1: Key Generation Algorithm of `Bouncy Castle` Implementation.

1. The object `field` of the class `GF2mField` represents the field which the Goppa Code will be defined on. This class implements operations with elements from the finite field $\mathbb{F}_{2^m} = \mathbb{F}_2[\gamma]$, where γ is a root of an irreducible polynomial with degree m. Each field element e has a polynomial basis representation, i.e. it is represented by a different binary polynomial of degree smaller than m, i.e. $e = e_{m-1}\gamma^{m-1} + \ldots + e_1\gamma + e_0$. All operations are defined only for extension fields with $1 < m < 32$. For the representation of field elements, a map $f : \mathbb{F}_{2^m} \to \mathbb{F}_2^m$ is used, where integers have the binary representation. For example, $\gamma^7 + \gamma^3 + \gamma + 1$ is represented by the integer $(00...0010001011) = 139$, which is stored in a variable `int`.
2. The object `gp` of the class `PolynomialGF2mSmallM` represents the irreducible Goppa polynomial $g(z)$ of degree t and whose coefficients are random elements taken from the field $\mathbb{F}_{2^m}$. It is stored as an `int` array $[g_0, g_1, \ldots, g_t]$, where g_i is the i-th coefficient of the polynomial.
3. The method `GoppaCode.createCanonicalCheckMatrix(field, gp)` will compute the parity check $\mathbf{H}$ for the Goppa code $\Gamma(\mathcal{L}, g(z))$, where $\mathcal{L}$ is the set of field elements represented by the set of integers $\{0, 1, 2, 3, \ldots, n-1\}$ with $n = 2^m$.

   In order to compute $\mathbf{H} = \mathbf{C}\cdot\mathbf{X}\cdot\mathbf{Y}$, the method makes use of an algorithm that follows directly from Equation (7.5). It then obtains a $(t \times n)$ `int` matrix `hArray`, where each entry is an element in $\mathbb{F}_{2^m}$. This matrix is then transformed into a $(t \cdot m) \times n$ matrix with entries in $\mathbb{F}_2$ and stored as a $(t \cdot m) \times \frac{(n+31)}{32}$ `int` array,[3] which is given as a parameter to create the object `h` of the class `GF2Matrix`. This class `GF2Matrix` implements some operations with matrices over $\mathbb{F}_2$.
4. The method `GoppaCode.computeSystematicForm(h, random)` will compute two matrices $\mathbf{A}$, $\mathbf{M}$ and a random permutation $\mathbf{P}_1$ such that $\mathbf{A}^{-1} \cdot \mathbf{H} \cdot \mathbf{P}_1 = (\mathbf{I} \mid \mathbf{M})$.

   In order to create the tuple $(\mathbf{A}, \mathbf{M}, \mathbf{P}_1)$, the method proceeds as follows. Since $\mathbf{H}$ is a $(t \cdot m) \times n$ matrix over $\mathbb{F}_2$, the method generates a random $n \times n$ permutation matrix $\mathbf{P}_1$, and then calculates $\mathbf{H} \cdot \mathbf{P}_1 = (\mathbf{A} \mid \mathbf{B})$, where $\mathbf{A}$ is a $(t \cdot m) \times (t \cdot m)$ binary matrix and $\mathbf{B}$ a $(t \cdot m) \times (n - t \cdot m)$ binary matrix. If $\mathbf{A}$ is invertible, then $\mathbf{A}^{-1} \cdot \mathbf{H} \cdot \mathbf{P}_1 = (\mathbf{I}_{t \cdot m} \mid \mathbf{M})$, where $I_{t \cdot m}$ is the $t \cdot m$ identity matrix and $\mathbf{M}$ is a $(t \cdot m) \times (n - t \cdot m)$ matrix. Otherwise, it picks another fresh, random permutation $\mathbf{P}_1$ and continues until it has found $\mathbf{A}$. It finally returns $(\mathbf{A}, \mathbf{M}, \mathbf{P}_1)$.

   The matrices $\mathbf{A}$, $\mathbf{M}$ are represented as objects of the class `GF2Matrix`, while the

[3] Each entry of the matrix `hArray` occupies only m out of 32 bits, so after packing all the entries of a row into a long bit string, this may be stored in only $\frac{(n+31)}{32}$ `int` entries.

permutation $\mathbf{P}_1$ is represented as an object of the class `Permutation`. This latter class implements permutations on the set $\{0, 1 \ldots, n-1\}$, for some given $n > 0$, as sequences containing each number l, $0 \leq l < n$, once and only once (n-permutations). Therefore, a sequence is stored internally as an n `int` array. Finally, the tuple $(\mathbf{A}, \mathbf{M}, \mathbf{P}_1)$ is represented by an object of the class `MaMaPe`.

Once the object `mmp` of the class `MaMaPe` has been created, the instruction `GF2Matrix shortH = mmp.getSecondMatrix()` simply retrieves $\mathbf{M}$ and assigns it to the object `shortH`. In a similar way, the instruction `Permutation p1 = mmp.getPermutation()` retrieves $\mathbf{P}_1$ and assigns it to the object `p1`.

5. The two following instructions follow from the observation that a parity check matrix can be used to construct the generator matrix for a code (and vice versa). If the parity check matrix $\mathbf{H}$ for a code C is in systematic form, i.e. $\mathbf{H} = (\mathbf{I}_k \mid \mathbf{M})$, then the generator matrix for C is $\mathbf{G} = (-\mathbf{M}^t \mid \mathbf{I}_{n-k})$, where $\mathbf{M}^t$ is the transpose of the matrix $\mathbf{M}$ [38]. Hence

   - `GF2Matrix shortG = (GF2Matrix)shortH.computeTranspose()` simply calculates $\mathbf{M}^t$ and assigns it to the object `shortG`.
   - `GF2Matrix gPrime = shortG.extendLeftCompactForm()` simply forms the matrix $\mathbf{G}' = (\mathbf{M}^t \mid I)$ and assigns it to the object `gPrime`.[4]

6. The instruction `int k = shortG.getNumRows()` simply obtains the number of rows of $\mathbf{G}'$, which is equal to the dimension of the code.
7. `GF2Matrix.createRandomRegularMatrixAndItsInverse(k, random)` will output a tuple $(\mathbf{S}, \mathbf{S}^{-1})$, where $\mathbf{S}$ is a $k \times k$ matrix with entries in $\mathbb{F}_2$ and $\mathbf{S}^{-1}$ is the inverse.
8. The instruction `Permutation p2 = new Permutation(n, random)` will create a random n-permutation $\mathbf{P}_2$ and assign it to the object `p2`.
9. The following two instructions will calculate $\mathbf{G} = \mathbf{S} \cdot \mathbf{G}' \cdot \mathbf{P}_2$ and assign it to the object `g`.
10. Finally both the public key and private key are created. The object `pubKey` will contain the tuple `(n,t,g)` while the object `privKey` will store

    `(n,k,field,gp,sInv,p1,p2,h)`,

    where

[4] The minus signs are not needed for fields of characteristic 2, i.e., $\mathbb{F}_{2^m}$.

- `n` is an `int` variable that holds the value of the length of the code.
- `t` is an `int` variable that holds the value of the error correction capability of the code.
- `k` is an `int` variable that holds the value of the dimension of the code.
- `g` is an object of the class `GF2Matrix`, which internally has a $k \times \frac{(n+31)}{32}$ `int` array that represents the $k \times n$ binary generator matrix $\mathbf{G}$.
- `field` is an object of the class `GF2mField`, which represents the underlying finite field. An object of this class contains the `int` variable `polynomial` that holds the value of the irreducible binary polynomial used for this field (value of `fieldPoly`).
- `gp` is an object of the class `PolynomialGF2mSmallM`, which represents the irreducible Goppa polynomial $g(z)$. This object internally has a $(t+1)$ `int` array to hold the coefficients of the polynomial. Each entry represents a field element.
- `sInv` is an object of the class `GF2Matrix`, which internally has a $k \times \frac{(k+31)}{32}$ `int` array that represents the $k \times k$ random binary non-singular matrix $\mathbf{S}^{-1}$.
- `p1` and `p2` are objects of the class `Permutation`. Each internally holds an n `int` array to store the values of the corresponding sequence of n integers; `p1` is used to generate the systematic check matrix, while `p2` is used to compute the public generator matrix.
- `h` is an object of the class `GF2Matrix`. It internally has a $(t \cdot m) \times \frac{(n+31)}{32}$ `int` array that holds the $t \cdot m \times n$ parity check matrix $\mathbf{H}$ of the code. Note that this object `h` is not given as a parameter when the object `privKey` is initialised, but instead `h` is created again internally by the same instruction used at step (3).

7.5 Mounting Cold Boot Attacks

In this section, we will explore algorithms that might recover the private key of this cryptosystem for the `Bouncy Castle` implementation in the cold boot attack setting.

7.5.1 Cold Boot Attack Model

We continue to make the assumptions outlined in Section 2.3. In particular, we assume that an adversary knows the values t, m, k, $n = 2^m$ and the integer value `fieldPoly` that represents the binary irreducible polynomial used for constructing the field $\mathbb{F}_{2^m}$. This as-

sumption is plausible since `fieldPoly` is created by a deterministic algorithm that receives the value of m when the McEliece parameters (an object of the class `McElieceParameters`) are being created. However, it is also possible to set this value and in such case it is very likely to be public. Additionally, the adversary obtains a noisy version of:

1. The $(t+1)$ `int` array `coefficients` which holds the coefficients of the Goppa polynomial (member of the object `gp`).
2. The n `int` array $\texttt{perm}_{p_1}$ which represents the n-permutation $\mathbf{P}_1$ (member of the object $\texttt{p}_1$).
3. The n `int` array $\texttt{perm}_{p_2}$ which represents the n-permutation $\mathbf{P}_2$ (member of the object $\texttt{p}_2$).
4. The $k \times \frac{(k+31)}{32}$ `int` array $\texttt{M}_s$ which holds the elements of the matrix $\mathbf{S}^{-1}$ (member of the object `sInv`).
5. The $(t \cdot m) \times \frac{(n+31)}{32}$ `int` array $\texttt{M}_h$ which holds the parity check matrix of the code (member of the object `h`).

So the attacker's goal is to recover the following components of the private key

$$(\texttt{coefficients}, \texttt{perm}_{p_1}, \texttt{perm}_{p_2}, \texttt{M}_s, \texttt{M}_h).$$

7.5.2 Key Recovery

A direct consequence of the manner in which the arrays $\texttt{M}_s$ and $\texttt{M}_h$ are created (look at step 3 of the key generation algorithm) is that any entry of these two arrays is dense, i.e. any entry can hold any possible integer value in the range $[-2^{31}, 2^{31}-1]$. Therefore, the attacker may not be able to exploit the noisy version of those entries. However, the attacker may be able to take advantage of the noisy version of the other arrays' entries. Recall that each entry of the array `coefficients` holds an integer value that represents a field element. Since this integer value is in the range $[0, 2^m - 1]$, it will be only stored in the first m of the 32 possible slots with the remaining slots filled with zeros. Similarly, each entry of the arrays $\texttt{perm}_{p_1}$ and $\texttt{perm}_{p_2}$ will only store integer values in the range $[0, n-1]$ with $n = 2^m$. Therefore, the attacker may create algorithms to generate high scoring and valid array candidates for those three arrays.

7.5.3 Key Recovery Algorithm

Let us assume we have an algorithm $\mathbb{A}$ that generates an array candidate of the form $[\mathtt{g}, \mathtt{p}_1, \mathtt{p}_2]$, where $\mathtt{g}, \mathtt{p}_1, \mathtt{p}_2$ are high scoring and valid array candidates for `coefficients`, $\mathtt{perm}_{p_1}$ and $\mathtt{perm}_{p_2}$ respectively. A valid array candidate for `coefficients` means that such an array represents an irreducible polynomial over $\mathbb{F}_{2^m}$. On the other hand, a valid array candidate for $\mathtt{perm}_{p_1}$ ($\mathtt{perm}_{p_2}$) means such an array represents a permutation on $\{0, 1, 2, \ldots, n-1\}$.

We next present a key recovery algorithm.

1. Call the algorithm $\mathbb{A}$ to generate an array candidate $[\mathtt{g}^{(i)}, \mathtt{p}_1^{(i)}, \mathtt{p}_2^{(i)}]$
2. Since $\mathtt{g}^{(i)}$ is a valid array candidate for `coefficients`, $\mathtt{g}^{(i)}$ represents an irreducible polynomial $g^{(i)}(z)$ of degree t that can be used to create the canonical check matrix $\mathbf{H}^{(i)}$ for the Goppa code $\Gamma^{(i)}(\mathcal{L}, g^{(i)}(z))$, where $\mathcal{L}$ is the set of field elements represented by the set of integers $\{0, 1, 2, 3, \ldots, n-1\}$ with $n = 2^m$, as done at step (3) of the key generation algorithm.
3. Once $\mathbf{H}^{(i)}$ is created, the attacker gets the permutation $\mathbf{P}_1^{(i)}$ from $\mathtt{p}_1^{(i)}$ and then computes $\mathbf{H}^{(i)}\mathbf{P}_1^{(i)} = (\mathbf{A}^{(i)} \mid \mathbf{B}^{(i)})$, where $\mathbf{A}^{(i)}$ is a $(t \cdot m) \times (t \cdot m)$ binary matrix and $\mathbf{B}^{(i)}$ a $(t \cdot m) \times (n - t \cdot m)$ binary matrix. If $\mathbf{A}^{(i)}$ is invertible, then the attacker computes $(\mathbf{A}^{(i)})^{-1} \cdot \mathbf{H}^{(i)} \cdot \mathbf{P}_1^{(i)} = (\mathbf{I}_{t \cdot m} \mid \mathbf{M}^{(i)})$, where $\mathbf{I}_{t \cdot m}$ is the $t \cdot m$ identity matrix and $\mathbf{M}^{(i)}$ is a $(t \cdot m) \times (n - t \cdot m)$ matrix. Hence the attacker gets $\mathbf{M}^{(i)}$, which can be then used to get the generation matrix $\mathbf{G}'^{(i)} = ((\mathbf{M}^{(i)})^t \mid \mathbf{I}_k)$ for this code, as done at step (5) of the key generation algorithm. Otherwise if $\mathbf{A}^{(i)}$ is not invertible, go to (1).
4. The attacker now computes $\mathbf{R}^{(i)} = \mathbf{G} \cdot (\mathbf{P}_2^{(i)})^{-1}$, where $\mathbf{G}$ is the public key and $\mathbf{P}_2^{(i)}$ is the permutation obtained from $\mathtt{p}_2^{(i)}$. Note that $\mathbf{R}^{(i)}$ can be computed by calculating $\mathbf{R}^{(i)} = \mathbf{G} \cdot (\mathbf{P}_2^{(i)})^t$, since the inverse of a permutation $\mathbf{P}$ is its transpose, i.e. $\mathbf{P}^{-1} = \mathbf{P}^t$. The attacker now can try to solve the equation $\mathbf{R}^{(i)} = \mathbf{S}^{(i)} \cdot \mathbf{G}'^{(i)}$, where

$$\mathbf{R}^{(i)} = \begin{pmatrix} r_{11}^{(i)} & r_{12}^{(i)} & r_{13}^{(i)} & \dots & r_{1n}^{(i)} \\ r_{21}^{(i)} & r_{22}^{(i)} & r_{23}^{(i)} & \dots & r_{2n}^{(i)} \\ r_{31}^{(i)} & r_{32}^{(i)} & r_{33}^{(i)} & \dots & r_{3n}^{(i)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{k1}^{(i)} & r_{k2}^{(i)} & r_{k3}^{(i)} & \dots & r_{kn}^{(i)} \end{pmatrix}, \mathbf{S}^{(i)} = \begin{pmatrix} s_{11}^{(i)} & s_{12}^{(i)} & s_{13}^{(i)} & \dots & s_{1k}^{(i)} \\ s_{21}^{(i)} & s_{22}^{(i)} & s_{23}^{(i)} & \dots & s_{2k}^{(i)} \\ s_{31}^{(i)} & s_{32}^{(i)} & s_{33}^{(i)} & \dots & s_{3k}^{(i)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{k1}^{(i)} & s_{k2}^{(i)} & s_{k3}^{(i)} & \dots & s_{kk}^{(i)} \end{pmatrix},$$

$$\mathbf{G'}^{(i)} = \begin{pmatrix} g_{11}^{\prime(i)} & \dots & g_{1(n-k)}^{\prime(i)} & 1 & 0 & 0 & \dots & 0 \\ g_{21}^{\prime(i)} & \dots & g_{2(n-k)}^{\prime(i)} & 0 & 1 & 0 & \dots & 0 \\ g_{31}^{\prime(i)} & \dots & g_{3(n-k)}^{\prime(i)} & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots & & \ddots & & \vdots \\ g_{k1}^{\prime(i)} & \dots & g_{k(n-k)}^{\prime(i)} & 0 & 0 & 0 & \dots & 1 \end{pmatrix}.$$

From $\mathbf{R}^{(i)} = \mathbf{S}^{(i)} \cdot \mathbf{G'}^{(i)}$, it follows that

$$r_{\hat{i}j}^{(i)} = \sum_{r=1}^{k} s_{\hat{i}r}^{(i)} g_{rj}^{\prime(i)} \text{ for } 1 \leq \hat{i} \leq k, 1 \leq j \leq n-k. \tag{7.15}$$

and

$$r_{\hat{i}(n-k+j)}^{(i)} = s_{\hat{i}j}^{(i)}, \text{ for } 1 \leq \hat{i} \leq k, 1 \leq j \leq k. \tag{7.16}$$

Therefore, from Equation (7.16), the attacker can get possible values for the entries of $\mathbf{S}^{(i)}$, and can then verify all of those values by substituting them in each of the expressions of Equation (7.15) and checking if each of the equations is satisfied. This step is efficiently done by extracting the sub-matrix of $\mathbf{R}^{(i)}$ consisting of the rightmost k columns and then multiplying it by $\mathbf{G'}^{(i)}$ while simultaneously checking if the partial results are equal to the corresponding entries of $\mathbf{R}^{(i)}$. If the check finishes successfully, then the attacker finds a candidate for the private key

$$(n, k, g^{(i)}(z), (\mathbf{S}^{(i)})^{-1}, \mathbf{P}_1^{(i)}, \mathbf{P}_2^{(i)}, \mathbf{H}^{(i)}).$$

Otherwise, go to step (1).

7.5.4 Constructing Array Candidates

In this section, we will show how we can make use of the key recovery strategy introduced in Section 4.4.1 to produce array candidates of the form $[\mathtt{g}, \mathtt{p}_1, \mathtt{p}_2]$, where $\mathtt{g}, \mathtt{p}_1, \mathtt{p}_2$ are high scoring and valid array candidates for the arrays `coefficients`, $\mathtt{perm}_{p_1}$ and $\mathtt{perm}_{p_2}$.

First the algorithm will receive the arrays $\mathtt{coefficients}'$, $\mathtt{perm}'_{p_1}$ and $\mathtt{perm}'_{p_2}$ as inputs. The arrays $\mathtt{coefficients}'$, $\mathtt{perm}'_{p_1}$ and $\mathtt{perm}'_{p_2}$ are the noisy versions of `coefficients`, $\mathtt{perm}_{p_1}$ and $\mathtt{perm}_{p_2}$ respectively. The attacker then constructs the lists of chunk candidates L_g, L_{P_1}, L_{P_2} from $\mathtt{coefficients}'$, $\mathtt{perm}'_{p_1}$ and $\mathtt{perm}'_{p_2}$ respectively and then combines them by picking a candidate from each list. Note that the attacker also may present these three lists as inputs to a key enumeration algorithm, regarding the lists L_g, L_{P_1}, L_{P_2} as sets of high scoring and valid array candidates for the arrays `coefficients`, $\mathtt{perm}_{p_1}$ and $\mathtt{perm}_{p_2}$. Note that the cost of our algorithm is given by $\mathcal{O}(|L_g| \cdot |L_{P_1}| \cdot |L_{P_2}|)$.

7.5.4.1 Constructing List of Candidates for Irreducible Polynomials

Here we show how to make use of the key recovery technique introduced in Section 4.4.1 to construct high scoring candidates for the array `coefficients`.

The attacker has access to a noisy version of `coefficients`, and so the attacker regards it as `r` and sets a chunk to be an `int` entry. Recall that each entry of the array `coefficients` holds an integer value that represents a field element. Since this integer value is in the range $[0, 2^m - 1]$, it will be only stored in the first m of the 32 possible slots with the remaining slots filled with zeros. Therefore, the candidates for any chunk are the integers in the set $\{0, 1, 2, 3, \ldots, 2^m - 1\}$.

Furthermore, the attacker represents `r` as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j} consecutive chunks, with $n_{b^j} > 0$. Therefore,

$$\mathtt{r} = \mathtt{b}^0 || \mathtt{b}^1 || \cdots || \mathtt{b}^{n_b - 1},$$

where

$$\mathtt{b}^j = \mathtt{r}^{i_j} || \mathtt{r}^{i_j+1} || \cdots || \mathtt{r}^{i_j+n_{b^j}-1},$$

for $0 \leq j < n_b$ and some $0 \leq i_j < t+1$.

In order to produce array candidates, the algorithm then proceeds as follows:

`Phase I`

1. For each chunk $\mathtt{r}^i$, $0 \leq i < t+1$, the attacker uses Equation (2.1) to compute log-likelihood scores for each candidate $\mathtt{c}^i$ for the chunk, viz.

$$\mathcal{L}[\mathtt{c}^i; \mathtt{r}^i] := n^i_{00} \log(1-\alpha) + n^i_{01} \log \alpha + n^i_{10} \log \beta + n^i_{11} \log(1-\beta),$$

   where the n^i_{ab} values count occurrences of bits across the i-th chunks, $\mathtt{r}^i$, $\mathtt{c}^i$. So the attacker obtains a list of chunk candidates with 2^m entries for the chunk.
2. For each block $\mathtt{b}^j$, $0 \leq j < n_b$, the attacker presents the n_{b^j} lists corresponding to the n_{b^j} chunks in the block $\mathtt{b}^j$ as inputs to OKEA to produce a list with the M_j highest scoring chunk candidates for the block, L_{b^j}.

Once `Phase I` is completed, the attacker will obtain n_b lists of chunk candidates and proceed as follows.

`Phase II`

1. The attacker presents the n_b lists as inputs to a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block $\mathtt{b}^j$. This instance will generate high scoring array candidates $\mathtt{g}$, where $\mathtt{g}$ is a $(t+1)$ `int` array. For each array candidate, the attacker checks if $\mathtt{g}$ conforms to being an irreducible polynomial over $\mathbb{F}_{2^m}$. If so, then $\mathtt{g}$ is a high scoring and valid array candidate and is inserted into L_g.

With regard to the check whether an $(t+1)$ `int` array $\mathtt{g}$ represents an irreducible polynomial over $\mathbb{F}_{2^m}$ or not, we use a standard algorithm described in [66] and implemented in the method `isIrreducible` of the class `PolynomialGF2mSmallM`.

7.5.4.2 Constructing List of Candidates for a 2^m-permutation

Here we show how to make use of the key recovery technique introduced in Section 4.4.1 to construct high scoring candidates for both the array $\texttt{perm}_{p_1}$ and the array $\texttt{perm}_{p_2}$.

The attacker now receives the noisy version of the array $\texttt{perm}_{p_1}(\texttt{perm}_{p_2})$ and regards it as $\texttt{r}$. The attacker then sets a chunk to be an `int` entry and its candidates to be the set of integers in the range between 0 and $n-1$, where $n = 2^m$. Recall the array $\texttt{perm}_{p_1}$ ($\texttt{perm}_{p_2}$) has 2^m entries, each of which stores an integer value in the range $[0, 2^m - 1]$. Furthermore, the attacker represents $\texttt{r}$ as a concatenation of n_b blocks, where each block consists of the concatenation of n_{b^j} consecutive chunks, with $n_{b^j} > 0$. Therefore,

$$\texttt{r} = \texttt{b}^0||\texttt{b}^1||\cdots||\texttt{b}^{n_b-1},$$

where

$$\texttt{b}^j = \texttt{r}^{i_j}||\texttt{r}^{i_j+1}||\cdots||\texttt{r}^{i_j+n_{b^j}-1},$$

for $0 \leq j < n_b$ and some $0 \leq i_j < 2^m$.

In order to produce array candidates, the algorithm then proceeds as follows:

`Phase I`

1. For each chunk $\texttt{r}^i$, $0 \leq i < 2^m$, the attacker uses Equation (2.1) to compute log-likelihood scores for each candidate $\texttt{c}^i$ for the chunk, viz.

$$\mathcal{L}[\texttt{c}^i; \texttt{r}^i] := n^i_{00}\log(1-\alpha) + n^i_{01}\log\alpha + n^i_{10}\log\beta + n^i_{11}\log(1-\beta),$$

   where the n^i_{ab} values count occurrences of bits across the i-th chunks, $\texttt{r}^i$, $\texttt{c}^i$. So the attacker obtains a list of chunk candidates with 2^m entries for the chunk.
2. For each block $\texttt{b}^j$, $0 \leq j < n_b$, the attacker presents the n_{b^j} lists corresponding to the n_{b^j} chunks in the block $\texttt{b}^j$ as inputs to OKEA to produce a list with the M_j highest scoring chunk candidates for the block, L_{b^j}.

Once `Phase I` is completed, the attacker will obtain n_b lists of chunk candidates and proceed as follows.

`Phase II`

1. The attacker presents the n_b lists as inputs to a key enumeration algorithm, regarding each list L_{b^j} as a set of candidates for the block $\mathtt{b}^j$. This instance will generate high scoring array candidates `p`, where `p` is a 2^m `int` array. For each array candidate, the attacker checks if `p` conforms to being a permutation on $\{0, 1, 2, \ldots, 2^m - 1\}$. If so, then `p` is a high scoring and valid array candidate and is inserted into $L_{P_1}(L_{P_2})$.

To check whether a 2^m `int` array `p` represents a permutation on $\{0, 1, 2, \ldots, 2^m - 1\}$ or not, the algorithm first sets a 2^m `boolean` array `B` to have each entry value `false` and then it iterates through $o = 0, 1, \ldots, 2^m - 1$ while checking if `B`[`p`[o]] equals `true`. If so, the algorithm stops and outputs `false`, meaning `p` does not represent a permutation on $\{0, 1, 2, \ldots, 2^m - 1\}$. Otherwise, it sets `B`[`p`[o]] to have the value `true` and continues operating. This algorithm is implemented in the method `isPermutation` of the class `Permutation`.

7.5.5 Parallelisation

The key recovery algorithm introduced in Section 7.5.3 can be parallelised if the algorithm $\mathbb{A}$ is parallelisable. Indeed, if $\mathbb{A}$ is parallelisable, it follows that we can have different instances $\mathbb{I}_i$ of $\mathbb{A}$, each of which generates high scoring and valid array candidates, hence we can run instances of the key recovery algorithm in parallel, each of which has an instance $\mathbb{I}_i$. We use the algorithm described in Section 7.5.4 as $\mathbb{A}$, which is parallelisable.

7.6 Experimental Evaluation

In this section, we will show some results from running the key recovery algorithm to find a private key for a particular parameter set. In particular, we will mainly focus on its success rate.

7.6.1 Implementation

We implement our key recovery algorithm in Java to take advantage of the implemented algorithms previously and other related algorithms provided by the `Bouncy Castle` im-

plementation.

7.6.2 Setup

We run our key recovery algorithm for an ad hoc set of parameters. In particular, $m = 9$ and $t = 31$, and $n = 512 = 2^9$. Besides, the integer value `fieldPoly` representing the irreducible binary polynomial defining $\mathbb{F}_{2^9}$ is set to 515, i.e. it is 1000000011 if seen as a bit string and represents the polynomial $z^9 + z + 1$. Therefore, the private key (once generated) consists of :

1. The $(t + 1)$ `int` array `coefficients` which holds the coefficients of the Goppa polynomial (member of the object `gp`).
2. The n `int` array $\texttt{perm}_{p_1}$ which represents the permutation $\mathbf{P}_1$ (member of the object $\texttt{p}_1$).
3. The n `int` array $\texttt{perm}_{p_2}$ which represents the permutation $\mathbf{P}_2$ (member of the object $\texttt{p}_2$).
4. The $k \times \frac{(k+31)}{32}$ `int` array $\texttt{M}_s$ which holds the elements of the matrix $\mathbf{S}^{-1}$ (member of the object `sInv`), where $k = 233$.
5. The $(t \cdot m) \times \frac{(n+31)}{32}$ `int` array $\texttt{M}_h$ which holds the check matrix of the code (member of the object `h`).

Recall that the attacker has access to a noisy version of the $(t+1)$ `int` array `coefficients`, a noisy version of the n `int` array $\texttt{perm}_{p_1}$ and a noisy version of the n `int` array $\texttt{perm}_{p_2}$.

As for producing array candidates with size 32 as candidates for `coefficients` from its noisy version, we set a chunk to be an `int` entry (32 bits) and set each block to be a consecutive sequence of 4 chunks, hence resulting in $n_b = 8 = 32/4$ blocks. Additionally, we set the set of candidates per chunk to be $\{0, 1, 2, \ldots, 511\}$ and the number of candidates to be generated for each block, $\texttt{blsize}_1$, to assume the values in the set $\{2^9, 2^{10}\}$. Therefore, any list output by `Phase I` has `blsize` entries. Once `Phase I` finishes, the resulting block lists are given as inputs to an instance of the key enumeration algorithm described in Section 5.6.2.3.[5] This instance will compute an interval of the form $[max - \delta, max]$ such that the number of candidates N_1 whose scores lie in the interval is at least $\texttt{lsize}_1$ but

[5] We use this algorithm because of its efficiency

the difference $N_1 - \texttt{lsize}_1$ is the smallest, where $\texttt{lsize}_1$ may assume the values in the set $\{2^{30}, 2^{35}, 2^{40}\}$. Hence, after `Phase II` has completed, we have $|L_g| \leq N_1$.

With regard to creating array candidates with size 512 as candidates for $\texttt{perm}_{p_1}$ ($\texttt{perm}_{p_2}$) from its noisy version, we set a chunk to be an `int` entry (32 bits) and set each block to be a sequence of 64 chunks, hence resulting in $n_b = 8 = 512/64$ blocks. Also, we set the set of candidates per chunk to be $\{0, 1, 2, \ldots, 511\}$ and the number of candidates to be generated for each block, $\texttt{blsize}_2$ ($\texttt{blsize}_3$), to assume the values in the set $\{2^9, 2^{10}\}$. Therefore, any list output by `Phase I` algorithm has $\texttt{blsize}_2$ ($\texttt{blsize}_3$) entries. Once `Phase I` finishes, the resulting block lists are given as inputs to an instance of the key enumeration algorithm described in Section 5.6.2.3. This instance will compute an interval of the form $[max - \delta, max]$ such that the number of candidates N_2 (N_3) whose scores lie in the interval is at least $\texttt{lsize}_2$ ($\texttt{lsize}_3$) but the difference $N_2 - \texttt{lsize}_2$ ($N_3 - \texttt{lsize}_3$) is the smallest. The parameter $\texttt{lsize}_2$ ($\texttt{lsize}_3$) may assume the values in the set $\{2^{30}, 2^{35}, 2^{40}\}$. Hence, after `Phase II` has completed, we have $|L_{P_1}| \leq N_2$ ($|L_{P_2}| \leq N_3$).

We next study the success rate of our algorithm.

7.6.3 Success Rate

For our algorithm to find the private key successfully, the correct components of it have to be contained in the lists L_g, L_{P_1}, L_{P_2} respectively. We use this observation to estimate the corresponding probabilities p_1, p_2, p_3, and therefore we may calculate $p_{success} = p_1 \cdot p_2 \cdot p_3$, for our selected parameter set. We considered data points $(\alpha, \beta) \in \{0.001\} \times \{0.001, 0.002, 0.003, 0.004, 0.005\}$, $\texttt{blsize}_o \in \{2^9, 2^{10}\}$, $\texttt{lsize}_o \in \{2^{30}, 2^{35}, 2^{40}\}$, $1 \leq o \leq 3$, and the corresponding full enumeration in `Phase II`. Note that given a data point (α, β), a same value for both $\texttt{lsize}_2$ and $\texttt{lsize}_3$, and a same value for both $\texttt{blsize}_2$ and $\texttt{blsize}_3$, it is expected that $p_2 \approx p_3$, because we use the same algorithm for generating both L_{P_1} and L_{P_2} (both lists contain candidates for 512-permutations).

To estimate these values, we have an auxiliary algorithm that first generates a fresh private key and perturbs each component of it according to α, β. This auxiliary algorithm then runs a tweaked key recovery algorithm $\mathbb{A}'$ for `coefficients` with the corresponding parameters, then runs it for $\texttt{perm}_{p_1}$ with the corresponding parameters. We run this auxiliary algorithm 100 times in our experiments.

Data Points		$\texttt{lsize}_1$				$\texttt{lsize}_2$			
α	β	2^{30}	2^{35}	2^{40}	2^{72}	2^{30}	2^{35}	2^{40}	2^{72}
0.001	0.001	1	1	1	1	0.19	0.26	0.28	0.34
0.001	0.002	1	1	1	1	0.02	0.09	0.1	0.11
0.001	0.003	1	1	1	1	0.01	0.01	0.02	0.04
0.001	0.004	1	1	1	1	0	0	0	0.02
0.001	0.005	1	1	1	1	0	0	0	0.01

Table 7.1: Values for p_1 and p_2 for the ad hoc parameter set, with $\texttt{blsize}_1 = \texttt{blsize}_2 = 2^9$.

In a run of algorithm $\mathbb{A}'$, it first receives the original `int` array `orginal`,[6] the corresponding noisy version `noisy`, the set $\{2^{30}, 2^{35}, 2^{40}\}$, the corresponding array `blimits`, which contains the indices of the last chunks of each block, and the corresponding value for `blsize`. This algorithm then partitions `noisy` into n_b blocks using `blimits` and run `Phase I`, hence producing the lists of chunk candidates $L_{b^j}, 0 \leq j < n_b$, each having `blsize` entries. It then proceeds to split the original array `orginal` into n_b blocks $\texttt{orginal}^0||\texttt{orginal}^1||\cdots||\texttt{orginal}^{n_b-1}$ using `blimits` and checks if the sub-arrays $\texttt{orginal}^j$ are contained in the lists L_{b^j} while accumulating their chunk scores. If the check does not succeed, it exits. Or else, the algorithm then counts a success for the full enumeration, since if `Phase II` is run completely, the correct array candidate will be output and inserted into the corresponding list. It then continues and, for each value `lsize` in the set $\{2^{30}, 2^{35}, 2^{40}\}$, it proceeds to compute an interval $[max - \delta, max]$ (exploiting the counting method of the key enumeration algorithm described in Section 5.6.2.3) and verify if the computed total score of the original array lies in the computed interval. If so, it means that if `Phase II` is run over the computed interval, the correct array candidate will be output and inserted into the corresponding list (therefore counting a success for the particular value of `lsize`).

Table 7.1 shows the results for the ad hoc parameter set, when $\texttt{blsize}_1 = 2^9$, $\texttt{blsize}_2 = 2^9$. On the other hand, Table 7.2 shows the results for the ad hoc parameter set, when $\texttt{blsize}_1 = 2^{10}$, $\texttt{blsize}_2 = 2^{10}$. If we set, $\texttt{blsize}_o = 2^9$, $\alpha = 0.001, \beta = 0.001, \texttt{lsize}_o = 2^{40}$, $1 \leq o \leq 3$, for example, then $p_{success} = (1) \cdot (0.28) \cdot (0.28) = 0.0784$ and the cost of

[6] Either `coefficients` or $\texttt{perm}_{p_1}$.

Data Points		$\texttt{lsize}_1$				$\texttt{lsize}_2$			
α	β	2^{30}	2^{35}	2^{40}	2^{80}	2^{30}	2^{35}	2^{40}	2^{80}
0.001	0.001	1	1	1	1	0.19	0.27	0.28	0.38
0.001	0.002	1	1	1	1	0.04	0.09	0.11	0.18
0.001	0.003	1	1	1	1	0	0.01	0.05	0.11
0.001	0.004	1	1	1	1	0	0.01	0.01	0.03
0.001	0.005	1	1	1	1	0	0	0	0.01

Table 7.2: Values for p_1 and p_2 for the ad hoc parameter set, with $\texttt{blsize}_1 = \texttt{blsize}_2 = 2^{10}$.

our key recovery algorithm would be at most 2^{120} operations. Recall that the lists L_g, L_{P_1}, L_{P_2} would have at most 2^{40} entries, since only valid candidates are included in any of the lists. Additionally, note that if $\texttt{lsize}_1$ were set to 2^{30}, the cost of our key recovery algorithm would be reduced to at most 2^{110}.

Furthermore, we observe that this recovery algorithm manages to tolerate very small values for both α and β. We believe that this behaviour is a result of the inherent design of this scheme, regardless of its implementation. This is because both permutations, $\mathbf{P}_1$ and $\mathbf{P}_2$, are chosen randomly during the key generation algorithm. Besides, their entries are integers in the set $\{0, 1, 2, \ldots, 511\}$.

Let us consider the default values 11 and 50 for m and t respectively. In such a case, $n = 2048$, $k = 1498$, and the integer value `fieldPoly` representing the irreducible binary polynomial defining $\mathbb{F}_{2^{11}}$ is set to 2053, i.e. it is 100000000101 if seen as a bit string and represents the polynomial $z^{11} + z^2 + 1$. So our algorithm will have to recover an irreducible polynomial stored in an `int` array of 51 entries, where each entry may assume any value in the set $\{0, 1, 2, \cdots, 2047\}$. Moreover, it will have to recover both the permutation $\mathbf{P}_1$ and the permutation $\mathbf{P}_2$. Each permutation is defined on the set $\{0, 1, 2, \cdots, 2047\}$ and therefore stored in an array of 2047 entries as a sequence containing each number l, $0 \leq l < 2048$, once and only once.

In this case, we expect that the probability $p_{success}$ for our key recovery algorithm to recover the correct private key be very small, even if the values for both α and β are very

small. In particular, we expect our key recovery algorithm to struggle with recovering the permutations $\mathbf{P}_1$ and $\mathbf{P}_2$, as we observed it did for the ad hoc parameter set. Hence, finding better algorithms to recover both the permutation $\mathbf{P}_1$ and the permutation $\mathbf{P}_2$ might improve our key recovery algorithm, specially its success rate.

7.7 Chapter Conclusions

In this chapter, we initiated the study of cold boot attacks on the McEliece public key encryption scheme, a member of the family of asymmetric cryptographic primitives based on linear codes. Our evaluation focused on an existing McEliece implementation provided by the `Bouncy Castle` Java crypto library. We proposed a key recovery algorithm exploiting the algebraic structure of this scheme in the cold boot attack setting, and made some experiments with this algorithm. As a future research, it would be interesting to search for other approaches exploiting better the algebraic structure of this scheme. In particular, exploiting further the relation $\mathbf{H} \cdot \mathbf{P}_1 = (\mathbf{A} \mid \mathbf{B})$, where $\mathbf{A}$ is an invertible $(t \cdot m) \times (t \cdot m)$ binary matrix and $\mathbf{B}$ a $(t \cdot m) \times (n - t \cdot m)$ binary matrix, could lead to a better algorithm to recover $\mathbf{P}_1$. Similarly, exploiting further the relation $\mathbf{G} = \mathbf{S} \cdot \mathbf{G}' \cdot \mathbf{P}_2$ could lead to a better algorithm to recover $\mathbf{P}_2$.

Chapter 8

Concluding Remarks

In this thesis, we studied algorithms by which an adversary might reconstruct a private key of a particular cryptographic scheme when the adversary is able to obtain a leakage function of the private key. In practice such leakage function could be obtained, although with some effort on the attacker's part, by gathering data leaked from the physical effects caused by the functioning of the cryptographic scheme's implementation [46]. In particular, we studied the case when the adversary procures data from a computer's main memory via a cold boot attack [28]. In Chapter 2, we studied the cold boot attack setting, in which the attacker with physical access to a machine may recover cryptographic key information of a cryptographic scheme via this data remanence attack. However, due to physical effects on the computer's main memory, any data retrieved from the memory (after pinpointing the location of the data in it) will probably have random bit fluctuations, i.e. the data will be noisy. Therefore, the adversary's main task is then the mathematical problem of recovering the original key from a noisy version of that key with help of any extra public information of the cryptographic scheme. This is, the main focus of cold boot attacks is to develop algorithms for efficiently recovering the original key from a noisy version of that key for the cryptographic scheme, whilst exploring the limits of how much noise can be tolerated. In particular, we focused on that task for several post-quantum cryptographic schemes.

We first posed the problem of key recovery in a general way and established a connection between the key recovery problem and the key enumeration problem. The latter problem arises in the side-channel attack literature, where, for example, the attacker might procure scoring information for each byte of an AES key from a side-channel attack and then want to efficiently enumerate and test a large number of complete 16-byte candidates until the correct key is found. In Chapter 3, we then studied several algorithms to solve the

key enumeration problem, such as the optimal key enumeration algorithm (OKEA) and several other non-optimal key enumeration algorithms. Additionally, we proposed variants of some of them and made a comparison of all of them, highlighting their strengths and weaknesses. This chapter paved the way for later study of cold boot attacks on post-quantum cryptographic schemes.

We then initiated the study of cold boot attacks on the NTRU public key encryption scheme, likely to be an important candidate in NIST's ongoing post-quantum standardisation process. In Chapter 4, we evaluated this cryptographic scheme in the cold boot attack setting and laid the groundwork for the later study of other cryptographic schemes in the same broad family of schemes that operate over polynomial rings. We proposed a general key recovery strategy using key enumeration algorithms for the key recovery problem. We then adapted it to tackle the problem for various private key formats from two existing NTRU implementations, the `ntru-crypto` implementation and the `tbuktu/Bouncy Castle` Java implementation. We experimented with variants of this general key recovery strategy to explore their performance for a range of parameters, showing how algorithms developed for enumerating keys in side-channel attacks can be successfully applied to the problem. In particular, our key recovery algorithm was able to tolerate a noise level of $\alpha = 0.001$ and $\beta = 0.09$ for one of the studied formats when performing a 2^{40} enumeration.

We then continued our task of evaluating post-quantum cryptographic schemes in the cold boot attack setting and turned our attention to the study of cold boot attacks on the BLISS signature scheme, a member of the same broad family of schemes that operate over polynomial rings. In Chapter 5, we evaluated such signature scheme in this setting and proposed algorithms for the key recovery problem, with particular emphasis on an existing BLISS implementation provided by the `strongSwan` project. We made use of the general key recovery strategy developed in Chapter 4 as a core algorithm. Furthermore, we established a connection between the key recovery problem in this particular case and an instance of Learning with Errors Problem (LWE). Additionally, we explored other techniques to tackle this LWE instance and also showed a technique combining lattice techniques with key enumeration. We then experimented with one of the key recovery algorithms to explore its performance for a range of parameters and found that our key recovery algorithm was able to tolerate a noise level of $\alpha = 0.001$ and $\beta = 0.09$ for a parameter set when performing a 2^{40} enumeration. As a future work, it may be interesting to pursue the research line of developing new key recovery techniques by combining key

enumeration algorithms with other techniques for solving Bounded Distance Decoding, e.g. lattice enumeration [25]. Another possible direction for future works is exploring key recovery algorithms exploiting the extra information stored in memory, such as the NTT of the coefficients of the public polynomial $(2\mathbf{g} + \mathbf{1})/\mathbf{f}$.

We continued our evaluation of post-quantum cryptographic schemes in the cold boot attack setting, with emphasis on the Rainbow signature scheme, which is a member of the family of asymmetric cryptographic primitives based on multivariate polynomials over a finite field K. In Chapter 6, we evaluated such cryptographic scheme in this setting and proposed a time-memory trade-off algorithm for key recovery, focusing our evaluation on two existing Rainbow implementations: the `Reference` implementation and the `Bouncy Castle` implementation. We experimented with our algorithm and discovered that our algorithm is not able to tolerate larger values for both α, β. We believe this is a consequence of the intrinsic design of this signature scheme, irrespective of its implementation. Hence, as a future work, it would be interesting to explore other approaches exploiting better the algebraic structure of this scheme.

In Chapter 7, we evaluated the McEliece public key encryption scheme in the cold boot attack setting. This scheme is a member of the family of asymmetric cryptographic primitives based on linear codes. Our evaluation focused on an existing McEliece implementation: The `Bouncy Castle` implementation. We proposed a key recovery algorithm exploiting the algebraic structure of this scheme for this particular implementation and experimented with this algorithm for an ad hoc parameter set. As a future work, it would be interesting to explore other approaches exploiting further the algebraic structure of this scheme. In particular, we think that exploiting further the relation $\mathbf{H} \cdot \mathbf{P}_1 = (\mathbf{A} \mid \mathbf{B})$, where $\mathbf{A}$ is an invertible $(t \cdot m) \times (t \cdot m)$ binary matrix and $\mathbf{B}$ a $(t \cdot m) \times (n - t \cdot m)$ binary matrix, could lead to a better algorithm to recover $\mathbf{P}_1$. Likewise, exploiting further the relation $\mathbf{G} = \mathbf{S} \cdot \mathbf{G}' \cdot \mathbf{P}_2$ could lead to a better algorithm to recover $\mathbf{P}_2$.

According to our results, our key recovery algorithms for cryptographic schemes in the broad family of schemes that operate over polynomial rings, such as BLISS and NTRU, tolerated much more noise than for the other cryptographic schemes in the two families of cryptographic schemes. A plausible reason for this is that both NTRU and BLISS's private key formats consist of a few arrays to store the underlying polynomials, i.e., the number of components to recover and hence needed to re-construct the private key is few.

Additionally, when a particular component is partitioned into chunks, any of these chunks may contain much more data than required, in the sense that the number of bits used to store a small number of candidates is greater than required. This redundancy as well as the small number of candidates per chunk allowed our attacks to generate more "reliable" scores for the candidates per chunk (hence, making our algorithms to find the correct component after enumerating much fewer candidates). From an implementer's view, this may be mitigated by reducing the redundancy used to store the polynomials, i.e., by employing simple packing techniques by which several coefficients are packed together when storing the private key.

On the other hand, as seen, our key recovery algorithms for Rainbow signature scheme and McEliece encryption scheme did not tolerate high levels of noise, even though these algorithms further exploited the algebraic structure of the corresponding schemes. A plausible reason for this is that the private key in both schemes consists of several components, each of which is represented with arrays having both a considerable number of chunks and a considerable number of candidates per chunk. Therefore, the redundancy is decreased in these formats and the correct component's rank is too high within the search space. This makes our enumeration techniques not to be as successful as they are in other scenarios. However, the attacker might still have a chance of improving these results by exploiting even further the algebraic structure so as to decrease the number of components to recover or the number of candidates per chunks when trying to recover a particular component.

Bibliography

[1] R. M. Avanzi, S. Hoerder, D. Page and M. Tunstall. Side-Channel Attacks on the McEliece and Niederreiter Public-Key Cryptosystems. Cryptology ePrint Archive, Report 2010/479, 2010. `http://eprint.iacr.org/2010/479`.

[2] M. Albrecht and C. Cid. Cold boot key recovery by solving polynomial systems with noise. In J. Lopez and G. Tsudik, editors, *ACNS 11*, volume 6715 of *LNCS*, pages 57 - 72. Springer, Heidelberg, June 2011.

[3] M. R. Albrecht, E. Orsini, K. G. Paterson, G. Peer, and N. P. Smart. Tightly secure ring-LWE based key encapsulation with short ciphertexts. Cryptology ePrint Archive, Report 2017/354, 2017. `http://eprint.iacr.org/2017/354`.

[4] M. R. Albrecht, A. Deo, K. G. Paterson. Cold Boot Attacks on Ring and Module LWE Keys Under the NTT. TCHES, vol. 2018, no. 3, pp. 173-213, Aug. 2018.

[5] L. Babai. On Lovász lattice reduction and the nearest lattice point problem. In K. Mehlhorn, editor, *STACS 1985*, volume 182 of *LNCS*. Springer, Berlin, Heidelberg, 1984.

[6] D. J. Bernstein et al. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In W. Fischer , N. Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017*, volume 10529 of *LNCS*. Springer, Cham, 2017.

[7] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU prime. Cryptology ePrint Archive, Report 2016/461, 2016. `http://eprint.iacr.org/2016/461`.

[8] D. J. Bernstein, T. Lange and C. Peters. Attacking and Defending the McEliece Cryptosystem. In J. Buchmann, J. Ding, editors, *PQCrypto 2008*, volume 5299 of *LNCS*, Springer, Berlin, Heidelberg, 2008.

[9] C. Chen, J. Hoffstein, W. Whyte2 and Z. Zhang. NIST PQ Submission: NTRU-Encrypt A lattice based encryption algorithm. `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[10] D. Engelbert, R. Overbeck and A. Schmidt. A Summary of McEliece-Type Cryptosystems and their Security. Cryptology ePrint Archive, Report 2006/162, 2016. `https://eprint.iacr.org/2006/162`.

[11] A. Bogdanov, I. Kizhvatov, K. Manzoor, E. Tischhauser, and M. Witteman. Fast and memory-efficient key recovery in side-channel attacks. In O. Dunkelman and L. Keliher, editors, *SAC 2015*, volume 9566 of *LNCS*, pages 310 - 327. Springer, Heidelberg, Aug. 2016.

[12] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. CRYSTALS - kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. `http://eprint.iacr.org/2017/634`.

[13] G. Bruinderink, A. Hülsing, T. Lange, Y. Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In B. Gierlichs and A. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, Volume 9813 of *LNCS*, 2016.

[14] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium*, Volume 12, USENIX Association, Berkeley, CA, USA, Aug. 2003.

[15] B. Brumley and N. Tuveri. Remote Timing Attacks are Still Practical. Cryptology ePrint Archive, Report 2011/232, 2011. `https://eprint.iacr.org/2011/232`.

[16] J. A. Buchmann,F. Göpfert, R. Player and T. Wunderer. On the Hardness of LWE with Binary Error: Revisiting the Hybrid Lattice-Reduction and Meet-in-the-Middle Attack. In D. Pointcheval, A. Nitaj, T. Rachidi, editors, volume 9646 of *LNCS*, pages

311 - 327, Springer, 2017.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. Introduction to Algorithms, 3rd edition, The MIT Press, 2009.

[18] L. David and A. Wool. A bounded-space near-optimal key enumeration algorithm for multi-subkey side-channel attacks. In H. Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *LNCS*, pages 311 - 327. Springer, 2017.

[19] J. Ding and D. J. Schmidt. Rainbow, a New Multivariable Polynomial Signature Scheme. In J. Ioannidis, A. Keromytis and M. Yung, editors, *Applied Cryptography and Network Security ACNS 2005*, Volume 3531 of *LNCS* , pages 164 - 175, Springer, Berlin, 2005.

[20] J. Ding, M. Chen, A. Petzoldt, D. Schmidt and B. Yang. Rainbow - Algorithm Specification and Documentation. `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.

[21] J. Ding and D.J. Schmidt. Multivariable Public Key Crypto-Systems. Cryptology ePrint Archive, Report 2004/350, 2004. `http://eprint.iacr.org/2004/350`.

[22] L. Ducas, A. Durmus, T. Lepoint and V. Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In R. Canetti, J.A. Garay, editors, *Advances in Cryptology CRYPTO 2013*, volume 8042 of *LNCS*, pages 40 - 56, Springer, 2013.

[23] L. Ducas. Accelerating Bliss: the geometry of ternary polynomials. Cryptology ePrint Archive, Report 2014/874, 2014. `http://eprint.iacr.org/2014/874`.

[24] T. Espitau, P. Fouque, B. Gérard, and M. Tibouchi. Side-Channel Attacks on BLISS Lattice-Based Signatures: Exploiting Branch Tracing against strongSwan and Electromagnetic Emanations in Micro-controllers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security CCS'17*, pages 1857 - 1874, 2017.

[25] N. Gama, P. Q. Nguyen, and O Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, EUROCRYPT 2010, volume 6110 of LNCS, pages 257 - 278. Springer, Heidelberg, May 2010.

[26] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC'96)*. ACM, New York, NY, USA, 212-219, 1996.

[27] L. K. Grover. Quantum mechanics helps in searching for a needle in a haystack. Phys. Rev. Lett. 79(2), 325, 1997

[28] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In P. C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28 - August 1, 2008, San Jose, CA, USA*, pages 45 - 60. USENIX Association, 2008.

[29] W. Henecka, A. May, and A. Meurer. Correcting errors in RSA private keys. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 351 - 369. Springer, Heidelberg, Aug. 2010.

[30] N. Heninger and H. Shacham. Reconstructing RSA private keys from random key bits. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 1 - 17. Springer, Heidelberg, Aug. 2009.

[31] J. Hoffstein, N. Howgrave-Graham, J. Pipher, and W. Whyte. Practical lattice-based cryptography: NTRUEncrypt and NTRUSign. In P. Q. Nguyen and B. Vallée, editors, *The LLL Algorithm - Survey and Applications*, Information Security and Cryptography, pages 349 - 390. Springer, 2010.

[32] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *LNCS*, pages 267 - 288. Springer, 1998.

[33] J. Hoffstein, J. Pipher, and J. H. Silverman. An Introduction to Mathematical Cryp-

tography, 2 Edition. *Springer Publishing Company*, 2014.

[34] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman1, W. Whyte, and Z. Zhang Choosing Parameters for NTRUEncrypt. Cryptology ePrint Archive: Report 2015/708, 2015. `https://eprint.iacr.org/2015/708`.

[35] N. Howgrave-Graham. A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU. In: A. Menezes, editors, *CRYPTO 2007*, volume 4622 of *LNCS*. Springer, Heidelberg, Aug. 2007.

[36] N. Howgrave-Graham, J. H. Silverman and W. Whyte. A meet-in-the-middle attack on an NTRU private key. Technical report, NTRU Crypto-systems, June 2003.

[37] Z. Huang and D. Lin. A new method for solving polynomial systems with noise over $\mathbb{F}_2$ and its applications in cold boot key recovery. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 16 - 33. Springer, Heidelberg, Aug. 2013.

[38] W. Huffman and V. Pless. Fundamentals of Error-Correcting Codes. Cambridge University Press, 2003.

[39] A. Kamal and A. M. Youssef. Applications of SAT solvers to AES key recovery from decayed key schedule images. In R. Savola, M. Takesue, R. Falk, and M. Popescu, editors, *Fourth International Conference on Emerging Security Information Systems and Technologies,SECURWARE 2010, Venice, Italy, July 18-25, 2010*, pages 216 - 220. IEEE Computer Society, 2010.

[40] A. Kipnis,J. Patarin and L. Goubin. Unbalanced oil and vinegar signature schemes. In J. Stern, editor, *Proceedings of the 17th international conference on Theory and application of cryptographic techniques EUROCRYPT'99*, pages 206 - 222, Springer-Verlag, Heidelberg, 1999.

[41] A. Kipnis,A. Shamir. Cryptanalysis of the oil and vinegar signature scheme. In H. Krawczyk, editor, *Advances in Cryptology CRYPTO 1998*, volume 1462 of *LNCS*, Springer, 1998.

[42] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N Koblitz, editor, *Advances in Cryptology CRYPTO 1996*, volume 1109 of *LNCS*, Springer, Berlin, Heidelberg, 1996.

[43] D. E. Knuth. The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms. Addison Wesley Longman Publishing Co., Redwood City, CA, USA.

[44] H. T. Lee, H. Kim, Y. J. Baek, and J. H. Cheon. Correcting errors in private keys obtained from cold boot attacks. In H. Kim, editor, *ICISC 11*, volume 7259 of *LNCS*, pages 74 - 87. Springer, Heidelberg, Nov. / Dec. 2012.

[45] J. Longo, D.P. Martin, L. Mather, E. Oswald, B. Sach and M. Stam. How low can you go? Using side-channel data to enhance brute-force key recovery. Cryptology ePrint Archive: Report 2016/609, 2016. `https://eprint.iacr.org/2016/609`.

[46] K. Mai. Side Channel Attacks and Countermeasures. In M. Tehranipoor and C. Wang, editors, *Introduction to Hardware Security and Trust*, Springer, New York, NY, 2012.

[47] D. P. Martin, L. Mather, E. Oswald, and M. Stam. Characterisation and estimation of the key rank distribution in the context of side channel evaluations. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 548 - 572. Springer, Heidelberg, Dec. 2016.

[48] D. P. Martin, J. F. O'Connell, E. Oswald, and M. Stam. Counting keys in parallel after a side channel attack. In T. Iwata and J. H. Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 313 - 337. Springer, Heidelberg, Nov. / Dec. 2015.

[49] D. P. Martin, L. Mather and E. Oswald. Two Sides of the Same Coin: Counting and Enumerating Keys Post Side-Channel Attacks Revisited. In N. Smart, editors, *Topics in Cryptology – CT-RSA 2018*, Volume 10808 of *LNCS*, springer, 2018.

[50] D. P. Martin,A. Montanaro, E. Oswald and D. Shepherd. Quantum Key Search with Side Channel Advice. In C. Adams and J. Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, Volume 10719 of *LNCS*, springer, 2017.

[51] R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, pages 114 - 116, 1978.

[52] S. Mittal,M. S.Inukonda A survey of techniques for improving error-resilience of DRAM. in *Journal of Systems Architecture*, Volume 91, Pages 11-40,2018.

[53] H. Niederreiter. Knapsack Type Cryptosystems and Algebraic Coding Theory. Problems of Control and Information Theory, 1986.

[54] R. Overbeck and N. Sendrier. Code-based cryptography. In D. J. Bernstein,J. Buchmann, E. Dahmen, editors, *Post-Quantum Cryptography*, Springer, 2009.

[55] K. G. Paterson, A. Polychroniadou, and D. L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 386 - 403. Springer, Heidelberg, Dec. 2012.

[56] J. Patarin. The oil and vinegar signature scheme. In *Dagstuhl Workshop on Cryptography*, 1997.

[57] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, pages 918 - 924, 1978.

[58] P. Pessl, L. G. Bruinderink and Y. Yarom. To BLISS-B or not to be - Attacking strongSwan's Implementation of Post-Quantum Signatures. Cryptology ePrint Archive, Report 2017/490, 2017. `https://eprint.iacr.org/2017/490`.

[59] R. Player. Parameter selection in lattice-based cryptography. Doctoral Thesis, Royal Holloway, University of London, 2018.

[60] B. Poettering and D. L. Sibborn. Cold boot attacks in the discrete logarithm setting. In K. Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 449 - 465. Springer, Heidelberg, Apr. 2015.

[61] R. Poussier, F. Standaert and V. Grosso. Simple Key Enumeration (and Rank Estimation) Using Histograms: An Integrated Approach In B. Gierlichs,A. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 61 - 81. Springer, Heidelberg, Jun.

2016.

[62] KA. Shim, CM. Park and YJ. Baek. Lite-Rainbow: Lightweight Signature Schemes Based on Multivariate Quadratic Equations and Their Secure Implementations. In A Biryukov,V. Goyal, editors, Progress in Cryptology *INDOCRYPT 2015*, Volume 9462 of *LNCS*, Springer, 2015.

[63] F. Strenzke. Timing Attacks against the Syndrome Inversion in Code-based Cryptosystems. Cryptology ePrint Archive, Report 2011/683, 2011. `http://eprint.iacr.org/2011/683`.

[64] J.H. Silverman and W. Whyte. Timing Attacks on NTRUEncrypt Via Variation in the Number of Hash Calls. In M. Abe, editors, *Topics in Cryptology - CT-RSA 2007*, Volume 4377 of *LNCS*, Springer, 2007.

[65] N. Sendrier. Code-Based Cryptography: State of the Art and Perspectives. In *IEEE Security & Privacy*, volume 15, no. 4, pages 44 - 50, 2017.

[66] V. Shoup. A Computational Introduction to Number Theory and Algebra (2 ed.). Cambridge University Press, New York, 2009.

[67] A. Shamir, N. van Someren. Playing "Hide and Seek" with Stored Keys. In M. Franklin, editors, *Financial Cryptography FC 1999*, volume 1648 of *LNCS*, Springer, Berlin, Aug. 1999.

[68] N. Seshadri and C. W. Sundberg. List viterbi decoding algorithms with applications, IEEE Transactions on Communications, pages 313 - 323, 1994.

[69] N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.-X. Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In L. R. Knudsen and H. Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 390 - 406. Springer, Heidelberg, Aug. 2013.

[70] P. C. Van Oorschot and M J. Wiener. Parallel Collision Search with Cryptanalytic Applications. In *Journal of Cryptology*, pages 1 - 28, 1999.

[71] C. Van Vredendaal. Reduced memory meet-in-the-middle attack against the NTRU private key. *LMS Journal of Computation and Mathematics*, pages 43 - 57, 2016.

[72] T. Wunderer. Revisiting the Hybrid Attack: Improved Analysis and Refined Security Estimates. Cryptology ePrint Archive, Report 2016/733, 2016. `http://eprint.iacr.org/2016/733`.

[73] Y. Yarom and K. Falkner Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive: Report 2013/448, 2013. `https://eprint.iacr.org/2013/448`.

[74] Y. Yarom, D. Genkin and N Heninger CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. Cryptology ePrint Archive: Report 2016/224, 2016. `https://eprint.iacr.org/2016/224`.

[75] IEEE Std 1363.1-2008, IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices, 2008.

[76] NIST: Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`.